LEVEL *II*

SEMI-FORMAL DESCRIPTION OF
KVM/370 TRUSTED PROCESSES

D. H. THOMPSON

DTIC
S **ELECTE** **D**
JAN 5 1982

D

, 9 DECEMBER 1977

CLEARED
FOR OPEN PUBLICATION

DEC 11 1981   3

DIRECTORATE FOR FREEDOM OF INFORMATION
AND SECURITY REVIEW (OASD—PA)
DEPARTMENT OF DEFENSE

SDC-TM-6062/111/00

34.7

81 12 22 106

KVM/370
Trusted Processes
Semi-Formal Description

This document contains a semi-formal description of the trusted
processes of the kernelized VM/370 operating system. The formal
specification and a prose description of the five trusted
processes are contained in document TM-6062/101/00, "KVM/370
Formal Specification".

Operator Process
Semi-Formal Description

This section contains a semi-formal description of the Operator
Process of KVM/370.

Data Types

primitive types and structuring mechanisms:

　　　boolean　[unordered, two elements: true, false]
　　　string　[unbounded, predefined string of length zero: nil]
　　　integer subrange

　　　scalar　[ordered element list]
　　　list　　[of any type, predefined empty list: nil]
　　　set　　[of any type, predefined empry set: nil]
　　　record　[field list]

undefined types:

　　　ProcessName
　　　MessageId
　　, CommandName

ConsoleOutputStatus: scalar(
　　　Continuing,
　　　Idle)

ResponseStatus: scalar(
　　　NoResponse,
　　　Responded)

RequestCategory: scalar(
　　　OpRequest,
　　　ReadOpRequest,
　　　PrintOpMsg,
　　　MapUserId)

Answer:
　　　record
　　　　HMS: string
　　　　Text: string
　　　end

2

```
ResponseSlot:
        record
           Respondent: ProcessName
           Text: string
           State: ResponseStatus
        end


PendingRequest:
        record
           MsgId: MessageId
           Kind: RequestCategory
           Command: CommandName
           Responses: set of ResponseSlot
        end


LogLine:
        record
           Num: 1..99
           Line: string
        end
```

3

Data Structures

Answers: list of Answer

PendingRequests: set of PendingRequest

LogMessage: set of LogLine

CommandExpected: boolean

ConsoleOutputState: ConsoleOutputStatus

CurrentNkcps: set of ProcessName

4

Initial Conditions

Empty(Answers)
&
Empty(PendingRequests)
&
Empty(LogMessage)
&
CommandExpected
&
ConsoleOutputState = Idle
&
Empty(CurrentNkcps)

Invariant Assertions

```
for all (P1,P2:PendingRequest) in PendingRequests:
    (P1.MsgId = P2.MsgId => P1 = P2)
&
for all (P:PendingRequest) in PendingRequests:
    (for all (R1,R2:ResponseSlot) in P.Responses:
        (R1.Respondent = R2.Respondent => R1 = R2)
     &
     for some (R:ResponseSlot) in P.Responses:
        (R.State = NoResponse)
     &
     ~Empty(P.Responses]))
&
for all (L1,L2:LogLine) in LogMessage:
    (L1.Num = L2.Num => L1 = L2)    .
```

Global Macros / Functions

primitive macros functions:

        Head[list]
        Tail[list]
        Empty[list/set]
        Append[list/set,entry]
        Insert[list,entry]

(as yet) undefined macros / functions:

        ClockRead
        MsgName
        Destination
        DeviceType

```
MsgNet: process

        /* subdriver of OpProcess,
                handling messages from Network Process */


        given:  MsgId: MessageId
                Text: string

        entry:  just received message, Source = NetworkProcess

        action: if for some (P:PendingRequest) in PendingRequests:
                        P.MsgId = MsgId
                    then /* response to request */
                        case P.Kind: RequestCategory of
                            PrintOpMsg:
                                case MsgName[Text] of
                                    OpMsgPrinted:
                                        if Empty(Answers)
                                            then ConsoleOutputState <-
                                                        Idle
                                            else KernelCall[SendMessage(
                                                    PrintOpMsg[
                                                        Head(Answers)],
                                                    NetworkProcess)]
                                                ConsoleOutputState <-
                                                    Continuing
                                                Answers <-
                                                        Tail(Answers)
                                    end

                                OpHitAttn:
                                    CommandExpected <- true
                                    ConsoleOutputState <- Idle
                                    KernelCall[SendMessage(
                                            ReadOpRequest,
                                            NetworkProcess)]

                                other:
                                    error
                                end
```

8

```
ReadOpRequest:
    case MsgName[Text] of
        OpRequestRead:
            ProcessCommand[Text]
            if CommandExpected
                &
                ConsoleOutputState = Idle
                &
                ~Empty[Answers]
                then KernelCall[SendMessage(
                        PrintOpMsg[
                            Head[Answers]],
                        NetworkProcess)]
                    ConsoleOutputState <-
                            Continuing
                    Answers <-
                            Tail[Answers]
            end

        OpHitAttn:
            CommandExpected <- true
            ConsoleOutputState <- Idle
            KernelCall[SendMessage(
                    ReadOpRequest,
                    NetworkProcess)]

        other:
            error
    end

OpRequest:
    error on MsgName[Text] ~= ResponseToOpRequest
    ProcessResponse

other:
    error
end
else /* request */
    case MsgName[Text] of
        OpHitAttn:
            CommandExpected <- true
            ConsoleOutputState <- Idle
            KernelCall[SendMessage(
                    ReadOpRequest,
                    NetworkProcess)]

        OperatorRequest:
            Answers <- Append[Answers,
                        <HMS = ClockRead[],
                        Text = Text>]
```

9

```
                                other:
                                    error
                        end
            end
end MsgNet
```

MsgAuth: process

        /* subdriver of OpProcess,
                handling messages from the Authorization Process */

        given: MsgId: MessageId
                Text: string

        entry: just received message, Source = AuthProcess

        action: If for some (P:PendingRequest) in PendingRequests:
                        P.MsgId = MsgId
                    then /* response to request */
                        case P.Kind:RequestCategory of
                            OpRequest:
                                error on MsgName[Text]
                                    ~= ResponseToOpRequest
                                ProcessResponse

                            MapUserId:
                                error on MsgName[Text]
                                    ~= MappedUserId
                                case P.Command:CommandName of
                                    FORCE←USERID,
                                    MESSAGE←USERID,
                                    WARNING←USERID:
                                        OpCat3a

                                    INDICATE←USER,
                                    LOCK←USERID,
                                    QUERY←PRIORITY,
                                    SET←FAVORED,
                                    SET←RESERVED,
                                    SET←PRIORITY,
                                    UNLOCK←USERID,
                                    LOCATE←USERID:
                                        OpCat3b

                                    ACNT←USERIDS:
                                        OpCat7

                                    other:
                                        error
                            end
                        other:
                            error
                    end

11

```
else /* request */
    case MsgName[Text] of
        OperatorReques*:
            Answers <-  ppend[Answers,
                                <HMS = ClockRead[],
                                 Text = Text>]

        AddNkcp:
            Auth1

        DeleteNkcp:
            Auth2

        other:
            error
    end
end
end MsgAuth
```

12

MsgAcnt: process

/* subdriver of OpProcess,
        handling messages from the Accounting Process */

given: MsgId: MessageId
        Text: string

entry: just received message, Source = AcntProcess

action: if for some (P.PendingRequest) in PendingRequests:
                P.MsgId = MsgId
            then /* response to request */
                    error on ~(P.Kind = OpRequest
                            &
                            MsgName[Text] = ResponseToOpRequest)
                    ProcessResponse
            else /* request */
                    error on MsgName [Text] ~= OperatorRequest
                    Answers <- Append[Answer,<HMS = ClockRead[],
                                              Text = Text>]
        end
end MsgAcnt

13

MsgNkcp: process

```
        /* subdriver of OpProcess,
                handling messages from all Nkcps */

    given:  MsgId: MessageId
            Text: String
            Process: ProcessName

    entry:  just received message, Source = Process (an Nkcp)

    action: if for some (P:PendingRequest) in PendingRequests:
                    P.MsgId = MsgId
                then /* response to request */
                    error on (P.Kind ~= OpRequest)
                                or
                            (MsgName[Text] ~= ResponseToOpRequest)
                        ProcessResponse
                else /* request */
                    error on MsgName[Text] ~= OperatorRequest
                    Answers <- Append[Answers,<HMS = ClockRead[],
                                                Text = Text>]
            end
end MsgNkcp
```

14

```
macro ProcessCommand[Text:string] =

        if CommandExpected
           then case CommandName[Text] of
                    DIAL,
                    LOGON,
                    LOGOFF,
                    SET←RECORD, SET←MODE,
                    SLEEP,
                    UNLOCK←SYSTEM, UNLOCK←VIRT,
                    ATTACH←CHANNEL,
                    DETACH←CHANNEL,
                    DISCONN,
                    SAVESYS:
                       OpCat0

                    AUTOLOG,
                    INDICATE←IO, INDICATE←PAGING,
                    LOCK←SYSTEM,
                    MONITOR←NKCP,
                    NETWORK,
                    QUERY←PAGING, QUERY←SASSIST←NKCP,
                    QUERY←DASD, QUERY←TAPES
                    QUERY←LINES, QUERY←GRAF,
                    QUERY←SYSTEM, QUERY←USERS,
                    QUERY←NAMES,
                    QUERY←USERID, QUERY←UR,
                    QUERY←DUMP, QUERY←FILES,
                    QUERY←READER, QUERY←PRINTER,
                    QUERY←PUNCH, QUERY←HOLD,
                    SET←SASSIST←NKCP, SET←DUMP,
                    UNLOCK←SYSTEM,
                    BACKSPAC,
                    CHANGE,
                    DRAIN,
                    FLUSH,
                    FREE,
                    HOLD,
                    ORDER,
                    PURGE,
                    REPEAT,
                    SPACE,
                    START,
                    TRANSFER,
                    DISABLE,
                    ENABLE,
                    LOADBUF,
                    ACNT←PUNCH,
                    DCP,
                    DMCP,
                    STCP:
                       OpCat1
```

15

```
                    FORCE←ALL,
                    INDICATE←LOAD, INDICATE←QUEUES,
                    MESSAGE←ALL,
                    MONITOR←ALL,
                    QUERY←SASSIST←ALL, QUERY←TDSK,
                    SET←SASSIST←ALL,
                    WARNING←ALL,
                    ACNT←ALL:
                        OpCat2

                    FORCE←USERID,
                    INDICATE←USER,
                    LOCK←USERID,
                    MESSAGE←USERID,
                    QUERY←PRIORITY,
                    SET←FAVORED, SET←RESERVED,
                    SET←PRIORITY,
                    UNLOCK←USERID,
                    WARNING←USERID,
                    LOCATE←USERID,
                    ACNT←USERIDS:
                        OpCat6

                    QUERY←RADDR,
                    ATTACH←RADDR,
                    DETACH←RADDR,
                    VARY,
                    LOCATE←RADDR:
                        OpCat4

                    HALT,
                    QUERY←STORAGE:
                        OpCat5

                    QUERY←LOGMSG:
                        OpCat8a

                    SET←LOGMSG:
                        OpCat8b

                    QUERY←ALL:
                        OpCat9

                    SHUTDOWN:
                        OpCat10

                    other:
                        error   /* specification error */
                end
          else /* must be set logmsg */
                OpCat8c
      end
```

16

```
OpDriver: process

        case HowWeGotHere of
            ExternalInterrupt:
                case InterruptSubType of
                    Message:
                        case Source of
                            AuthProcess:        MsgAuth
                            URProcess:          MsgUR
                            Nkcp:               MsgNkcp
                            IOPagingScheduler:  MsgIOPS
                            IOScheduler:        MsgIOS
                            NetworkProcess:     MsgNet
                            DumpProcessor:      MsgDump
                            AcntProcess:        MsgAcnt
                            other:              /* anybody else talk
                                                   to OpProcess? */

                    end

                    other:
                        /* any other external interrupts? */
                end

            other:
                /* any other important interrupt classes? */
        end

        KernelCall[ReceiveInterrupts]
        KernelCall[ReleaseCPU]
end OpDriver
```

17

Entry / Exit Conditions

OpCat0:   Not legal

     commands:
           DIAL
           LOGON
           LOGOFF
           SET←RECORD
           SET←MODE
           SLEEP
           UNLOCK←SYSTEM
           UNLOCK←VIRT
           ATTACH←CHANNEL
           DETACH←CHANNEL
           DISCONN
           SAVESYS

    exit: N"Answers = Append[Answers,ErrorMessage]
          N"CommandExpected = true

          where
          ErrorMessage = <HMS = ClockRead[],
                       Text = "Not a legal command.">

OpCat1:   Single message sent
                 No mappings
                 Single response expected (unless marked by ☆)

        commands:
                AUTOLOG   QUERY←PAGING   ☆SET←DUMP
                INDICATE←IO   ←SASSIST←NKCP UNLOCK←SYSTEM
                      ←PAGING   ←DASD   BACKSPAC
                LOCK←SYSTEM   ←TAPES   CHANGE
                MONITOR←NKCP   ←LINES   DRAIN
                NETWORK   ←GRAF   FLUSH
                                 ←SYSTEM   FREE
                                 ←NAMES   HOLD
                                 ←USERS   ORDER
                                 ←USERID   PURGE
                                 ←UR   REPEAT
                                 ←DUMP   SPACE
                                 ←FILES   START
                                 ←READER   TRANSFER
                                 ←PRINTER   DISABLE
                                 ←PUNCH   ENABLE
                                 ←HOLD   LOADBUF
                          SET←SASSIST←NKCP ACNT←PUNCH
                                            DCP
                                            DMCP
                                            STCP

        exit: N"PendingRequests = Append(PendingRequests,Entry)
                 N"CommandExpected = true

                 where
                 Entry = <MsgId = new(OpMsg),
                          Kind = OpRequest,
                          Command = Command,
                          Responses = set:
                     (<Respondent = Destination(Command),
                      Text = nil,
                      State = NoResponse>)>

        KernelCalled(SendMessage(Destination(Command)))

OpCat2:   Multiple messages sent -- one to each NKCP
                        No mappings
                        Multiple responses expected (unless marked by *)

        commands:
                *FORCE←ALL
                 INDICATE←LOAD
                 INDICATE←QUEUES
                 MESSAGE←ALL
                 MONITOR←ALL
                 QUERY←SASSIST←ALL
                 QUERY←TOSK
                 SET←SASSIST←ALL
                *WARNING←ALL
                 ACNT←ALL

        exit: N"PendingRequests = Append(PendingRequests,Entry)
                N"CommandExpected = true

                where
                Entry = <MsgId = new(OpMsg),
                        Kind = OpRequest,
                        Command = Command,
                        Responses = set:(for all (N:ProcessName)
                                                in CurrentNkcps:
                                            <Respondent = N,
                                             Text = nil,
                                             State = NoResponse>)>

                for all (N:ProcessName) in CurrentNkcps:
                        KernelCalled(SendMessage(N))

20

OpCat3:   Single message sent
                       Mapping: user Id -> NKCP Id (already performed)
                       OpCat3a: No response expected (marked by *)
                       OpCat3b: Single response expected

          commands:
               *FORCE←USERID
                INDICATE←USER
                LOCK←USERID
               *MESSAGE←USERID
                QUERY←PRIORITY
                SET←FAVORED
                SET←RESERVED
                SET←PRIORITY
                UNLOCK←USERID
               *WARNING←USERID
                LOCATE←USERID

     given: Nkcps: set of ProcessName

     entry: for all (N1:ProcessName) in Nkcps:
                  for some (N2:ProcessName)
                                in Current:Nkcps:

                      N1 = N2

     exit: N"PendingRequests = Append(PendingRequests,Entry)
             N"CommandExpected = true

             where
             Entry = <MsgId = new(OpMsg),
                      Kind = OpRequest,
                      Command = Command,
                      Responses = set: (for all (N:ProcessName)
                                            in Nkcps:

                           <Respondent = N,
                            Text = nil,
                            State = NoResponse>}>

         for all (N:ProcessName) in Nkcps:
                 KernelCalled(SendMessage(N))

OpCat4:   Single message sent to:
                      either URProcess or AuthProcess
                 Mapping: raddr -> device type
                 Single response expected

          commands:
                 QUERY←RADDR
                 ATTACH←RADDR
                 DETACH←RADDR
                 VARY
                 LOCATE←RADDR

          given: Raddr: string

          exit: N"PendingRequests =
                        if DeviceType[Raddr] = UnitRecord
                           then Append[PendingRequests,Entry1]
                           else Append[PendingRequests,Entry2]
                        end
                  N"CommandExpected = true

                  where
                  Entry1 = <MsgId = new[OpMsg],
                            Kind = OpRequest,
                            Command = Command,
                            Responses = set:
                               [<Respondent = URProcess,
                                 Text = nil,
                                 State = NoResponse>] >
                  Entry2 = <Msgid = new[OpMsg],
                            Kind = OpRequest,
                            Command = Command,
                            Responses = set:
                               [<Respondent = AuthProcess,
                                 Text = nil,
                                 State = NoResponse>] >

                  if DeviceType[Raddr] = UnitRecord
                     then KernelCalled[SendMessage[URProcess]]
                     else KernelCalled[SendMessage[AuthProcess]]
                  end

22

OpCat5:   Kernel call
                     No message sent
                     Command answered

          commands:
                     HALT
                     QUERY-STORAGE

          exit: N"Answers = Append[Answers,Entry]
                  N"CommandExpected = true

                  KernelCalled[Command(KResponse)]

                  where
                  KResponse: string
                  Entry = <HMS = ClockRead[],
                           Text = KResponse>

OpCat6:   Single message to AuthProcess,
                    to perform user Id -> NKCP Id mapping function
                 Response expected

        commands:
                 all commands In Op categorias 3 and 7

        exit: N"PendingRequests = Append(PendingRequests,Entry)
                 N"CommandExpected = true

                 where
                 Entry = <MsgId = new(OpMsg),
                            Kind = MapUserId,
                            Comman' = Command,
                            Respc ..es = set:
                              (<Respondent = AuthProcess,
                               Text = nil,
                               State = NoResponse>)>

        KernelCalled(SendMessage(AuthProcess))

24

OpCat7:

Multiple messages to a subset of NKCPs
Response expected from each
Mapping: user id -> NKCP id (already performed)

command:
ACNT+USERIDS

given: Nkcps: set of ProcessName

entry: for all (N1:ProcessName) in Nkcps:
            for some (N2:ProcessName) in CurrentNkcps:
                N1 = N2

exit: N"PendingRequests = Append(PendingRequests,Entry)
        N"CommandExpected = true

        for all (N:ProcessName) in Nkcps:
            KernelCalled(SendMessage(N))

        where
        Entry = <MsgId = new(OpMsg),
                Kind = OpRequest,
                Command = Command,
                Responses: set:
                    (for all (N:ProcessName) in Nkcps:
                        <Respondent = N,
                         Text = nil,
                         State = NoResponse>)>

25

OpCat8:   LogMessage processing

       commands:
                QUERY←LOGMSG
                SET←LOGMSG

OpCat8a:
       QUERY←LOGMSG:
       exit: N"Answers = Append[Answers,Entry]
                N"CommandExpected = true

                where
                Entry = <HMS = ClockRead[],
                           Text = MakeString[LogMessage]>

OpCat8b:
       SET←LOGMSG:
       exit: N"Answers = Insert[Answers,Prompt]
                N"CommandExpected = false

                where
                Prompt = <HMS = nil,
                            Text = "LOGMSG:">

OpCat8c:
       LogMsg Line Received
       given: Line#: 1..99
               Line: string

       entry: CommandExpected = false

       exit: N"CommandExpected = true
        •   N"LogMessage =
                        if for some (L:LogLine) in LogMessage:
                              (L.Num = Line#)
                           then Append[Remove[LogMessage,L],Entry]
                           else Append[LogMessage,Entry]
                        end

                where
                Entry = <Num = Line#,
                            Line = Line>

OpCat9:   Multiple messages to:
                      AuthProcess and URProcess
                  Kernel call
                  No mappings
                  Multiple responses expected

    command:
          QUERY←ALL

    exit: N"PendingRequests = Append[PendingRequests,Entry]
             N"CommandExpected = true

             KernelCalled[Storage(KResponse)]
             KernelCalled[SendMessage(AuthProcess)]
             KernelCalled[SendMessage(URProcess)]

             where
             KResponse: String
             Entry = <MsgId = new[OpMsg],
                      Kind = OpRequest,
                      Command = Command,
                      Responses = set:
                          [<Respondent = Kernel,
                            Text = KResponse,
                            State = Responded>,
                           <Respondent = AuthProcess,
                            Text = nil,
                            State = NoResponse>,
                           <Respondent = URProcess,
                            Text = nil,
                            State = NoResponse>] >

27

OpCat10:   Multiple messages to:
                  AuthProcess, URProcess, and all NKCPs
                  Kernel call
                  No mappings
                  No responses expected

     command:
          SHUTDOWN

     exit: N"CommandExpected = true

             KernelCalled[Shutdown]
             KernelCalled[SendMessage(AuthProcess)]
             KernelCalled[SendMessage(URProcess)]
             KernelCalled[SendMessage(AcntProcess)]
             for all (N:ProcessName) in CurrentNkcps:
                 KernelCalled[SendMessage(N)]

28

Auth1:   Add Nkcp

given: Nkcp: ProcessName

entry: true

exit: N"CurrentNkcps = Append(CurrentNkcps,Nkcp)

Auth2: Purge Nkcp

     given: Nkcp: ProcessName

     entry: for some (N:ProcessName) in CurrentNkcps:
            N = Nkcp

     exit: N"CurrentNkcps = Remove(CurrentNkcps,Nkcp)

Unit Record Process
Semi-Formal Description

This section contains a semi-formal description of the Unit Record
Process of KVM/370.

Data Types

primitive types and structuring mechanisms:

        boolean [unordered, two elements: true, false]
        string   [unbounded, predefined string of length zero: nil]
        integer subrange

        scalar   (ordered element list]
        set   [of any type, predefined empty set: nil]
        record   [field list]
        union   [list of types or data structures]

undefined types:

        DeviceAddress
        HardwareStatus
        MessageId
        ProcessName

Class: scalar(
        Aye,
        Bee,
        Cee,
        All)

ResponseStatus: scalar(
        NoResponse,
        Responded)

RelDevRequestStatus: scalar(/* Relinquish Device Request Status */
        NoNeed,
        ShouldSend,
        Sent)

31

```
CommandName: scalar(
        QUERY+UR+ALL,
        QUERY+RADOR,
        QUERY+FILES+USERID, QUERY+FILES+ALL,
        QUERY+ROPRPU+USERID, QUERY+ROPRPU+SPOOLID,
        QUERY+ROPRPU+ALL,
        QUERY+HOLD,
        BACKSPAC,
        CHANGE+USERID, CHANGE+SYSTEM,
        DRAIN,
        FLUSH,
        FREE,
        HOLD,
        ORDER+USERID, ORDER+SYSTEM,
        PURGE+USERID, PURGE+SYSTEM,
        REPEAT,
        SPACE,
        TRANSFER,
        LOCATE)


RequestCategory: scalar(
        OpRequest,
        MapUserId,
        NeedNkcp,
        RelinquishDevice)


InputDeviceStatus: scalar(
        SecurityHeader,
        SecurityHeaderWaitForReady,
        AttachPending,
        AttachedToSpoolingProcess,
        Available,
        AttachedToUser,
        DetachPending,
        OffLine)


OutputDeviceStatus: scalar(
        SecurityHeader,
        SecurityHeaderWaitForReady,
        AttachedToSpoolingProcess,
        SecurityTrailer,
        SecurityTrailerWaitForReady,
        Available,
        AttachedToUser,
        DetachPending,
        OffLine,
        LoadbufPending,
        LoadbufWaitForReady)
```

32

```
TapeDriveStatus: scalar(
        Available,
        Attached,
        OffLine,
        DetachPending)


ActivityStatus: scalar(
        NotSpooling,
        Drained,
        Started,
        Draining)


SpoolId:
        record
          Process: ProcessName
          File: 0..999
        end


ODRequestStatus: scalar(
        Processing,
        WaitingForDevice)


OutputDeviceRequest:
        record
          Process: ProcessName
          RequestedClasses: set of Class
          AttachedDev: DeviceAddress
          State: ODRequestStatus
        end


TapeDriveEntry:
        record
          Raddr: DeviceAddress
          State: TapeDriveStatus
          AttachedProcess: ProcessName
        end


ResponseSlot:
        record
          Respondent: ProcessName
          Text: string
          State: ResponseStatus
        end
```

33

```
PendingRequest:
        record
           MsgId: MessageId
           Kind: RequestCategory
           Command: CommandName
           Responses: set of ResponseSlot
        end


ReaderEntry:
        record
           Raddr: DeviceAddress
           State: ActivityStatus
           CyclePosition: InputDeviceStatus
           AttachedProcess: ProcessName
           ClassesServedCurrently: set of Class
           ClassesServedNextCycle: set of Class
           ChannelStatusWord: HardwareStatus
           LineBuffer: string
        end


PrinterEntry:
        record
           Raddr: DeviceAddress
           State: ActivityStatus
           CyclePosition: OutputDeviceStatus
           AttachedProcess: ProcessName
           ClassesServedCurrently: set of Class
           ClassesServedNextCycle: set of Class
           RelinquishDeviceRequestState: RelDevRequestStatus
           ChannelStatusWord: HardwareStatus
        end


PunchEntry:
        record
           Raddr: DeviceAddress
           State: ActivityStatus
           CyclePosition: OutputDeviceStatus
           AttachedProcess: ProcessName
           ClassesServedCurrently: set of Class
           ClassesServedNextCycle: set of Class
           RelinquishDeviceRequestState: RelDevRequestStatus
           ChannelStatusWord: HardwareStatus
        end
```

34

```
NkcpEntry:
        record
            Process: ProcessName
            UsableReaders: set of DeviceAddress
            UsablePrinters: set of DeviceAddress
            UsablePunches: set of DeviceAddress
            UsableTapeDrives: set of DeviceAddress
        end

DeviceEntry: union of (
        ReaderEntry,
        PrinterEntry,
        PunchEntry)

OutputDeviceEntry: union of (
        PrinterEntry,
        PunchEntry)
```

Data Structures

ShuttingDown: boolean

Readers: set of ReaderEntry

Printers: set of PrinterEntry

Punches: set of PunchEntry

TapeDrives: set of TapeDriveEntry

PrinterSpoolRequests: set of OutputDeviceRequest

PunchSpoolRequests: set of OutputDeviceRequest

CurrentNkcps: set of NkcpEntry

PendingRequests: set of PendingRequest

Devices: union of (Readers, Printers, Punches)

36

(Initial Conditions

Empty(PendingRequests)
&
Empty(CurrentNkcps)
&
Empty(PrinterSpoolRequests)
&
Empty(PunchSpoolRequests)
&
(~ShuttingDown)
&
for all (R:ReaderEntry) in Readers:
    (ReaderInInitialState(R))
&
for all (Pr:PrinterEntry) in Printers:
    (PrinterInInitialState(Pr))
&
for all (Pu:PunchEntry) in Punches:
    (PunchInInitialState(Pu))
&
for all (T:TapeDriveEntry) in TapeDrives:
    (TapeDriveInInitialState(T))

(

```
macro ReaderInInitialState(R:ReaderEntry) =
        (R.State = Drained
        &
        R.CyclePosition = Available
        &
        R.AttachedProcess = URProcess
        &
        R.LineBuffer = nil
        &
        Empty(R.ClassesServedCurrently)
        &
        Empty(R.ClassesServedNextCycle))


macro PrinterInInitialState(P:PrinterEntry) =
        (P.State = Drained
        &
        P.CyclePosition = Available
        &
        P.AttachedProcess = URProcess
        &
        Empty(P.ClassesServedCurrently)
        &
        Empty(P.ClassesServedNextCycle)
        &
        P.RelinquishDeviceRequestState = NoNeed)


macro PunchInInitialState(P:PunchEntry) =
        (P.State = Drained
        &
        P.CyclePosition = Available
        &
        P.AttachedProcess = URProcess
        &
        Empty(P.ClassesServedCurrently)
        &
        Empty(P.ClassesServedNextCycle)
        &
        P.RelinquishDeviceRequestState = NoNeed)


macro TapeDriveInInitialState(T:TapeDriveEntry) =
        (T.State = Available
        &
        T.AttachedProcess = URProcess)
```

38

Invariant Assertions

```
for all (R1, R2:ReaderEntry) in Readers:
    R1.Raddr = R2.Raddr => R1 = R2

for all (Pr1, Pr2:PrinterEntry) in Printers:
    Pr1.Raddr = Pr2.Raddr => Pr1 = Pr2

for all (Pu1, Pu2:PunchEntry) in Punches:
    Pu1.Raddr = Pu2.Raddr => Pu1 = Pu2

for all (T1, T2:TapeDriveEntry) in TapeDrives:
    T1.Raddr = T2.Raddr => T1 = T2


for all (R:ReaderEntry) in Readers:
    (for all (Pr:PrinterEntry) in Printers:
        (R.Raddr ~= Pr.Raddr)
    &
    for all (Pu:PunchEntry) in Punches:
        (R.Raddr ~= Pu.Raddr)
    &
    for all (T:TapeDriveEntry) in TapeDrives:
        (R.Raddr ~= T.Raddr))
&
for all (Pr:PrinterEntry) in Printers:
    (for all (Pu:PunchEntry) in Punches:
        (Pr.Raddr ~= Pu.Raddr)
    &
    for all (T:TapeDriveEntry) in TapeDrives:
        (Pr.Raddr ~= T.Raddr))
&
for all (Pu:PunchEntry) in Punches:
    for all (T:TapeDriveEntry) in TapeDrives:
        Pu.Raddr ~= T.Raddr
```

39

```
for all (R:ReaderEntry) in Readers:
    (
    ((R.State = NotSpooling) =>
        R.CyclePosition inset (AttachPending,
                               DetachPending,
                               AttachedToUser,
                               OffLine))
    &
    ((R.State ~= NotSpooling) =>
        R.CyclePosition inset (SecurityHeader,
                               SecurityHeaderWaitForReady,
                               AttachPending,
                               AttachedToSpoolingProcess,
                               Available))
    &
    ((R.CyclePosition = Available) =>
        (R.ClassesServedCurrently = R.ClassesServedNextCycle
         &
         R.AttachedProcess = URProcess))
    )


for all (P:OutputDeviceEntry) in OutputDevices:
    ((P.State = NotSpooling =>
        P.CyclePosition inset (AttachPending,
                               DetachPending,
                               AttachedToUser,
                               OffLine))
    &
    (P.State ~= NotSpooling =>
        P.CyclePosition inset (SecurityHeader,
                               SecurityHeaderWaitForReady,
                               SecurityTrailer,
                               SecurityTrailerWaitForReady,
                               AttachedToSpoolingProcess,
                               Available))
    &
    (P.CyclePosition = Available =>
        (P.ClassesServedCurrently = P.ClassesServedNextCycle
         &
         P.AttachedProcess = URProcess))
    )


for all (DR:OutputDeviceRequest) in PrinterSpoolRequests:
    for some (N:NkcpEntry) in CurrentNkcps:
        N.Process = DR.Process

for all (DR:OutputDeviceRequest) in PunchSpoolRequests:
    for some (N:NkcpEntry) in CurrentNkcps:
        N.Process = DR.Process
```

```
for all (N1,N2:NkcpEntry) in CurrentNkcps:
    (N1.Process = N2.Process => N1 = N2)


for all (P1,P2:PendingRequest) in PendingRequests:
    (P1.Msgid = P2.Msgid => P1 = P2)
&
for all (P:PendingRequest) in PendingRequests:
    (for all (R1,R2:ResponseSlot) in P.Responses:
        (R1.Respondent = R2.Respondent => R1 = R2)
     &
     for some (R:ResponseSlot) in P.Responses:
        (R.State = NoRespone)
     &
     ~Empty(P.Responses))


for all (N:NkcpEntry) in CurrentNkcps:
    ((for all (D:DeviceAddress) in N.UsableReaders:
            for some (R:ReaderEntry) in Readers:
                R.Raddr = D)
     &
     (for all (D:DeviceAddress) in N.UsablePrinters:
            for some (Pr:PrinterEntry) in Printers:
                Pr.Raddr = D)
     &
     (for all (D:DeviceAddress) in N.UsablePunches:
            for some (Pu:PunchEntry) in Punches:
                Pu.Raddr = D)
     &
     (for all (D:DeviceAddress) in N.UsableTapeDrives:
            for some (T:TapeDriveEntry) in TapeDrives:
                T.Raddr = D))
```

41

Legality Checks upheld in Driver:

```
IOInterrupt(Raddr) =>
    for some (D:DeviceEntry) in Devices:
        (D.Raddr = Raddr
         &
         D.CyclePosition inset (Available, SecurityHeader,
                                SecurityHeaderWaitForReady,
                                SecurityTrailer,
                                SecurityTrailerWaitForReady))
```

Global Macros / Functions

primitive macros / functions:

        Append[set,entry]
        Remove[set,entry]
        Empty[set,entry]

undefined macros / functions:

        KernelCalled

```
macro ReInitializeInputSpoolDevice[R:ReaderEntry] =
        if R.State = Draining
            then R.State <- Drained
                KernelCall[SendMessage(Drained[R.Raddr],OpProcess)]
        end
        R.CyclePosition <- Available
        R.ClassesServedCurrently <- R.ClassesServedNextCycle
        R.AttachedProcess <- URProcess

macro ClassesMatch[S1,S2:set of Class] =
        for some (C1:Class) in S1:
            for some (C2:Class) in S2:
                C1 = C2
```

43

RD: process

```
        /* subdriver of URProcess,
                handling IO interrupts on readers */

        given:  R:ReaderEntry in Readers

        entry:  just received interrupt on R.Raddr

        action: /* pay attention to only those interrupts
                        that we care about */

        error on R.State inset (NotSpooling,
                                        Drained)
                or
                R.CyclePosition inset (
                                AttachPending,
                                AttachedToSpoolingProcess,
                                AttachedToUser,
                                DetachPending,
                                OffLine)
        case R.CyclePosition:InputDeviceStatus of
                SecurityHeader:
                        if R.ChannelStatusWord.UnitCheck
                            then RD2b
                            else RD2a
                        end

                SecurityHeaderWaitForReady:
                        if R.ChannelStatusWord.UnitCheck
                            then RD2b
                            else RD2c
                        end

                Available:
                        if ShuttingDown
                            then KernelCall[SendMessage(
                                    PhysicallyPurgeDeck[R.Raddr],
                                    OpProcess)]
                            else RD1
                        end
                end
        end

end RD
```

44

PR: process

```
/* subdriver of URProcess,
          handling IO interrupts on printers */

given:  P:PrinterEntry in Printers

entry:  just received interrupt on P.Raddr

action: error on P.State inset (NotSpooling,
                                Drained)
              or
              P.CyclePosition inset (
                                AttachedToSpoolingProcess,
                                AttachedToUser,
                                DetachPending,
                                OffLine)

        case P.CyclePosition: OutputDeviceStatus of
        SecurityHeader:
            if P.ChannelStatusWord.UnitCheck
               then PR3b
               else PR3a
            end

        SecurityHeaderWaitForReady:
            if P.ChannelStatusWord.UnitCheck
               then PR3b
               else PR3c
            end

        Available:
            if (~Empty(PrinterSpoolRequests))
               &
               P.State = Started
               &
               for some (DR:OutputDeviceRequest)
                          in PrinterSpoolRequests:
                  (DR.State = WaitingForDevice
                   &
                   ClassesMatch[
                           P.ClassesServedCurrently,
                           DR.RequestedClasses])
               then PR2
            end
```

45

```
SecurityTrailer:
    if P.ChannelStatusWord.UnitCheck
        then PR5b
        else PR5a
    end

SecurityTrailerWaitForReady:
    if P.ChannelStatusWord.UnitCheck
        then PR5b
        else PR5c
    end

LoadbufPending:
    PR6a

LoadbufWaitForReady:
    PR6b
                                        end
end PR
```

MsgAuth: process

        /* subdriver of URProcess,
                handling messages from AuthProcess */

        given: MsgId: MessageId
                Text: string

        entry: just received message, Source = AuthProcess

        action: if for some (P:PendingRequest) in PendingRequests:
                        P.MsgId = MsgId
                    then /* response to request */
                        case P.Kind: RequestCategory of
                            NeedNkcp:
                                case MsgName[Text] of
                                    AddedNkcp:
                                        DetermineRaddr[Text]
                                        error on
                                            for all (R:ReaderEntry)
                                                in Readers:
                                                    (R.Raddr ~= Raddr)
                                        RD3a
                                        PendingRequests <- Remove[
                                                PendingRequests,P]

                                    CannotAddNkcp:
                                        DetermineRaddr[Text]
                                        error on
                                            for all (R:ReaderEntry)
                                                in Readers:
                                                    (R.Raddr ~= Raddr)
                                        RD3b
                                        PendingRequests <- Remove[
                                                PendingRequests,P]

                                    other:
                                        error
                    end

47

(

```
MapUserId:
    error on MsgName[Text] ~= UserIdMapped
    if Nkcp[Text] = nil
        then KernelCall[SendMessage(
                UnknownUserId[Text],
                OpProcess)]
            PendingRequests <- Remove[
                    PendingRequests,P]
        else KernelCall[SendMessage(
                OpCmd[Text],
                Nkcp[Text])]
    end

other:
    error
end
else /* message is a request */
    case MsgName[Text] of
        AddNkcp:
            AUTH1

        DeleteNkcp:
            AUTH2

        AttachDevice:
            DetermineRaddr[Text]
            case DeviceType[Raddr] of
                Reader,
                Printer,
                Punch:
                    AUTH3abc

                TapeDrive:
                    AUTH3d
            end

        other:
            error
    end
end
end MsgAuth
```

MsgOp: process

    /* subdriver of URProcess,
        handling messages from OpProcess */

    given: MsgId: MessageId
          Text: string

    entry: just received message, Source = OpProcess

    action: if for some (P:PendingRequest) in PendingRequests:
            P.MsgId = MsgId
        then /* response to request */
            /* URProcess currently doesn't make any
               requests of OpProcess: error */
          error
        else case MsgName[Text] of
              QUERY←UR,
              QUERY←ALL,
              QUERY←RADDR,
              QUERY←TAPES,
                OP0

              QUERY←READER←SPOOLID,
              QUERY←PRINTER←SPOOLID,
              QUERY←PUNCH←SPOOLID,
              CHANGE←SYSTEM←SPOOLID:
                OP1

              QUERY←FILES←ALL,
              QUERY←READER←ALL,
              QUERY←PRINTER←ALL,
              QUERY←PUNCH←ALL,
              QUERY←HOLD,
              CHANGE←SYSTEM←CLASS←ALL:
                OP2

              QUERY←FILES←USERID,
              QUERY←READER←USERID,
              QUERY←PRINTER←USERID,
              QUERY←PUNCH←USERID,
              CHANGE←USERID,
              FREE,
              HOLD,
              ORDER←USERID,
              PURGE←USERID:
                OP3

```
BACKSPAC,
FLUSH,
REPEAT:
    DetermineRaddr [Text]
    case DeviceType (Raddr) of
        Printer,
        Punch:
            OP4ab

        other:
            error
    end

SPACE:
    DetermineRaddr [Text]
    case DeviceType (Raddr) of
        Printer:
            OP5

        other:
            error
    end

ORDER←SYSTEM,
PURGE←SYSTEM:
    OP6

VARY←OFFLINE:
    DetermineRaddr [Text]
    case DeviceType (Raddr) of
        Reader,
        Printer,
        Punch:
            OP9abc

        TapeDrive:
            OP9d
    end
```

```
VARY-ONLINE:
    DetermineRaddr [Text]
    case DeviceType [Raddr] of
            Reader:
                OP10a

            Printer,
            Punch:
                OP10bc

            TapeDrive:
                OP10d
    end

ATTACH:
    DetermineRaddr [Text]
    case DeviceType [Raddr] of
            Reader,
            Printer,
            Punch:
                OP11abc

            TapeDrive:
                OP11d
    end

DETACH:
    DetermineRaddr [Text]
    case DeviceType [Raddr] of
            Reader,
            Printer,
            Punch:
                OP12abc

            TapeDrive:
                OP12d
    end

LOCATE:
    DetermineRaddr [Text]
    case DeviceType [Raddr] of
            Reader,
            Printer,
            Punch:
                OP7abc

            TapeDrive:
                OP7d
    end

SHUTDOWN:
    OP8
```

```
START:
    DetermineRaddr(Text)
    case DeviceType(Raddr) of
            Reader:
                OP15a

            Printer,
            Punch:
                OP15bc

            TapeDrive:
                error
    end

DRAIN:
    DetermineRaddr(Text)
    case DeviceType(Raddr) of
            Reader:
                OP14a

            Printer,
            Punch:
                OP14bc

            TapeDrive:
                error
    end

TRANSFER:
    OP16

LOADBUF:
    DetermineRaddr(Text)
    case DeviceType(Raddr) of
            Printer:
                OP13

            other:
                error
    end

    other:
        error
            end
        end
end MsgOp
```

MsgNkcp: process

                /* subdriver of URProcess,
                        handling messages from Nkcps */

        given: Process: ProcessName
                 MsgId: MessageId
                 Text: string

        entry: just received message, Source = Process

        action: if for some (P:PendingRequest) in PendingRequests:
                        P.MsgId = MsgId
                    then /* response to request */
                        case P.Kind:RequestCategory of
                            OpRequest:
                                error on
                                    MsgName[Text] ~= ResponseToOpRequest
                                ProcessResponse

                            RelinquishDevice:
                                error on MsgName[Text] ~= DetachDevice
                                DetermineRaddr[Text]
                                case DeviceType[Raddr] of
                                        Reader,
                                        Printer,
                                        Punch:
                                            NKCP1abc

                                        TapeDrive:
                                            NKCP1d
                                end

                            other:
                                error
                        end
                    else /* request */
                        case MsgName[Text] of
                            DetachSpoolDevice:
                                DetermineRaddr[Text]
                                case DeviceType[Raddr] of
                                        Reader:
                                            RD4

                                        Printer:
                                            PR4

                                        Punch:
                                            PU4

                                        TapeDrive:
                                            error
                                end

                                    53

```
DetachDevice:
    DetermineRaddr [Text]
    case DeviceType [Raddr] of
            Reader,
            Printer,
            Punch:
                NKCP2abc

            TapeDrive:
                NKCP2d
    end

NeedSpoolingDevice:
    DetermineRaddr [Text]
    case DeviceType [Raddr] of
            Printer:
                PR1

            Punch:
                PU1

            other:
                error
    end

other:
    error
                end
        end
end MsgNkcp
```

54

```
URDriver: process

        case HowWeGotHere of
            IOInterrupt:
                case DeviceType of
                    Reader:    RD
                    Printer:   PR
                    Punch:     PU
                    TapeDrive: error
                end

            ExternalInterrupt:
                case InterruptSubType of
                    Message:
                        case Source of
                            AuthProcess: MsgAuth
                            OpProcess:   MsgOp
                            Nkcp:        MsgNkcp
                            other:       /* anybody else talk
                                            with URProcess? */
                        end

                    other:
                        /* any other external interrupts? */
                end

            other:
                /* any other important interrupt classes? */
        end

        KernelCall[ReceiveInterrupts]
        KernelCall[ReleaseCPU]
end URDriver
```

55

RD1:   Unexpected IO interrupt signalling deck to be read

        given: Raddr: DeviceAddress

        entry: for some (R:ReaderEntry) in Readers:
                     (R.Raddr = Raddr
                       &
                      R.State = Started
                       &
                      R.CyclePosition = Available)
                  &
                  ~ ShuttingDown

        action: R.CyclePosition <- SecurityHeader
                R.LineBuffer <- nil
                KernelCall (RequestIO(R.Raddr) for input]

        exit: N"R.CyclePosition = SecurityHeader
                N"R.LineBuffer = nil

                KernelCalled(RequestIO(R.Raddr) for input]

R02a: Expected IO interrupt signalling security header read

        given: Raddr: DeviceAddress
                 Process: ProcessName

        entry: for some (R:ReaderEntry) in Readers:
                     (R.Raddr = Raddr
                      &
                      R.State ~= NotSpooling
                      &
                      R.CyclePosition = SecurityHeader
                      &
                      R.ChannelStatusWord.UnitCheck = false
                      &
                      R.LineBuffer ~= nil)

        action: if for some (N:NkcpEntry) in CurrentNkcps:
                        (N.Process = Process)
                    then if for some (D:DeviceAddress)
                                      in N.UsableReaders:
                               (D = R.Raddr)
                           then /* Attach Device */
                               KernelCall [GrantAccess(
                                       N.Process,R.Raddr)]
                               if OK
                                  then R.AttachedProcess <- N.Process
                                       R.CyclePosition
                                          <- AttachedToSpoolingProcess
                                       KernelCall [SendMessage(
                                          SpoolDeviceAttached[R.Raddr],
                                          R.AttachedProcess)]
                                  else /* Kernel did not grant access:
                                       something's wrong */
                                       KernelCall [SendMessage(
                                          PhysicallyPurgeDeck[R.Raddr],
                                          OpProcess)]
                                       ReInitializeInputSpoolCycle[R]
                           end
                    else /* Nkcp exists but
                         cannot use this reader */
                         KernelCall [SendMessage(
                                 PhysicallyPurgeDeck[R.Raddr],
                                 OpProcess)]
                         ReInitializeInputSpoolCycle[R]
                 end

57

```
                       else /* necessary Nkcp does not currently exist */
                           R.CyclePosition <- AttachPending
                           R.AttachedProcess <- Process
                           KernelCall[SendMessage(
                                   NeedNkcp[R.AttachedProcess,R.Raddr],
                                   AuthProcess)]
              end

     exit:    N"R.CyclePosition =
                      if for some (N:NkcpEntry) in CurrentNkcps:
                              N.Process = Process
                          then if for some (D:DeviceAddress)
                                                   in N.UsableReaders:
                                      (D = R.Raddr)
                                      &
                                      GrantedAccess
                                      then AttachedToSpoolingProcess
                                      else Available
                                  end
                          else AttachPending
                      end
              N"R.AttachedProcess =
                      if for some (N:NkcpEntry) in CurrentNkcps:
                              N.Process = Process
                          then if for some (D:DeviceAddress)
                                                   in N.UsableReaders:
                                      (D = R.Raddr)
                                      &
                                      GrantedAccess
                                      then N.Process
                                      else URProcess
                                  end
                          else Process
                      end
              N"R.State =
                      if for some (N:NkcpEntry) in CurrentNkcps:
                              (N.Process = Process)
                          &
                          (for all (D:DeviceAddress)
                                                   in N.UsableReaders:
                                  (D ~= R.Raddr)
                              or
                              ~ GrantedAccess)
                          &
                          R.State = Draining
                          then Drained
                          else R.State
                      end
```

58

```
N"R.ClassesServedCurrently =
        if for some (N:NkcpEntry) in CurrentNkcps:
                (N.Process = Process)
            &
            for all (D:DeviceAddress)
                                    in N.UsableReaders:
                (D ~= R.Raddr)
            &
            ~ GrantedAccess
            then R.ClassesServedNextCycle
            else R.ClassesServedCurrently
        end

if for some (N:NkcpEntry) in CurrentNkcps:
        (N.Process = Process)
    then if for some (D:DeviceAddress) in N.UsableReaders:
                (D = R.Raddr)
            then KernelCalled[GrantAccess]
                    if GrantedAccess
                        then KernelCalled[SendMessage(
                                        N.Process)]
                        else KernelCalled[SendMessage(
                                        OpProcess)]
                            if R.State = Draining
                                then KernelCalled[
                                        SendMessage(
                                            OpP    ess)]
                    end
            end
            else KernelCalled[SendMessage(OpProcess)]
                    if R.State = Draining
                        then KernelCalled[SendMessage(
                                        OpProcess)]
                    end
        end
    else KernelCalled[SendMessage(AuthProcess)]
end
```

RO2b: IO Error when attempting security header read

        given: Raddr: DeviceAddress

        entry: for some (R:ReaderEntry) in Readers:
                     (R.Raddr = Raddr
                      &
                      R.State ~= NotSpooling
                      &
                      (R.CyclePosition inset (SecurityHeader,
                                              SecurityHeaderWaitForReady))
                      &
                      R.ChannelStatusWord.UnitCheck = true)

        action: R.CyclePosition <- SecurityHeaderWaitForReady
                 KernelCall (SendMessage(InterventionRequired (R.Raddr),
                                         OpProcess))

        exit: N"R.CyclePosition = SecurityHeaderWaitForReady

                 KernelCalled (SendMessage(OpProcess))

R02c: Reader error cleared by operator (IO interrupt signalling
device ready)

        given: Raddr: DeviceAddress

        entry: for some (R:ReaderEntry) in Readers:
                     (R.Raddr = Raddr
                      &
                      R.State ~= NotSpooling
                      &
                      R.CyclePosition = SecurityHeaderWaitForReady
                      &
                      R.ChannelStatusWord.UnitCheck = false)

        action: R.CyclePosition <- SecurityHeader
                 R.LineBuffer <- nil
                 KernelCall(RequestIO(R.Raddr) for input]

        exit: N"R.CyclePosition = SecurityHeader
                 N"R.LineBuffer = nil

                 KernelCalled(RequestIO(R.Raddr) for input]

RO3a: AuthProcess message re Nkcp creation: added

        given: Raddr: DeviceAddress

        entry: for some (R:ReaderEntry) in Readers:
                    (R.Raddr = Raddr)
                &
                for some (N:NkcpEntry) in CurrentNkcps:
                    (N.Process = R.AttachedProcess)

        error on ~(R.State ~= NotSpooling
                &
                R.CyclePosition = AttachPending)

        action: if for some (D:DeviceAddress) in N.UsableReaders:
                    D = R.Raddr
                then /* attach process */
                    KernelCall [GrantAccess(N.Process,R.Raddr)]
                    if OK
                        then R.CyclePosition <-
                                    AttachedToSpoolingProcess
                            KernelCall [SendMessage(
                                    SpoolDeviceAttached [R.Raddr],
                                    N.Process)]
                        else /* Kernel did not grant access */
                            KernelCall [SendMessage(
                                    PhysicallyPurgeDeck [R.Raddr],
                                    OpProcess)]
                            KernelCall [SendMessage(
                                    PurgeIfAble,
                                    N.Process)]
                            ReInitializeInputSpoolCycle [R]
                    end
                else /* NKCP exists but cannot use this reader */
                    KernelCall [SendMessage(
                            PhysicallyPurgeDeck [R.Raddr],
                            OpProcess)]
                    KernelCall [SendMessage(
                            PurgeIfAble,
                            N.Process)]
                    ReInitializeInputSpoolCycle [R]
            end

```
exit:    N"R.CyclePosition =
                  if for some (D:DeviceAddress)
                                     in N.UsableReaders:
                       (D = R.Raddr)
                    &
                    GrantedAccess
                    then AttachedToSpoolingProcess
                    else Available
              end
         N"R.State =
                  if (for all (D:DeviceAddress)
                                     in N.UsableReaders:
                       (D ~= R.Raddr)
                      or
                      ~ GrantedAccess)
                    &
                    R.State = Draining
                    then Drained
                    else R.State
              end
         N"R.ClassesServedCurrently =
                  if for all (D:DeviceAddress)
                                     in N.UsableReaders:
                       (D ~= R.Raddr)
                    or
                    ~ GrantedAccess
                    then R.ClassesServedNextCycle
                    else R.ClassesServedCurrently
              end
         N"R.AttachedProcess =
                  if for all (D:DeviceAddress)
                                     in N.UsableReaders:
                       (D ~= R.Raddr)
                    or
                    ~ GrantedAccess
                    then URProcess
                    else R.AttachedProcess
                 end

         KernelCalled(SendMessage(N.Process))
         if for some (D:DeviceAddress) in N.UsableReaders:
              (D = R.Raddr)
            then KernelCalled(GrantAccess)
                 if ~ GrantedAccess
                    then KernelCalled(SendMessage(OpProcess))
                 end
            else KernelCalled(SendMessage(OpProcess))
         end
```

RO3b: AuthProcess message re Nkcp creation: cannot add

        given: Raddr: DeviceAddress

        entry: for some (R:ReaderEntry) in Readers:
                     R.Raddr = Raddr

        error on ~(R.State ~= NotSpooling
                   &
                   R.CyclePosition = AttachPending)

        action: KernelCall(SendMessage(
                     PhysicallyPurgeDeck[R.Raddr],
                     OpProcess))
                ReinitializeInputSpoolDevice[R]

        exit: N"R.State =
                        if R.State = Draining
                            then Drained
                            else R.State
                        end
                 N"R.CyclePosition = Available
                 N"R.ClassesServedCurrently = R.ClassesServedNextCycle
                 N"R.AttachedProcess = URProcess

                 KernelCalled(SendMessage(upProcess))
                 if R.State = Draining
                     then KernelCalled(SendMessage(OpProcess))
                 end

64

RO4:   Process releases reader after spooling

      given: Raddr: DeviceAddress
            RequestingProcess: ProcessName

      entry: for some (R:ReaderEntry) in Readers:
               R.Raddr = Raddr

      error on ~(R.State ~= NotSpooling
                        &
                  R.CyclePosition = AttachedtoSpoolingProcess
                        &
                  R.AttachedProcess = RequestingProcess)

      action: KernelCall[ReleaseDevice(R.AttachedProcess,R.Raddr)]
          if OK
             then ReInitializeInputSpoolDevice(R)
             else KernelCall[SendMessage(
                        DeviceNotReleased(R.Raddr),
                        R.AttachedProcess)]
                  KernelCalled[SendMEssage(
                        DeviceNotReleased[
                            R.Raddr,
                            R.AttachedProcess],
                      OpProcess)]
         end

65

```
exit:    N"R.State =
                    if DeviceReleased
                        &
                        R.State = Draining
                        then Drained
                        else R.State
                    end
            N"R.CyclePosition =
                    if DeviceReleased
                        then Available
                        else R.CyclePosition
                    end
            N"R.ClassesServedCurrently =
                    if DeviceReleased
                        then R.ClassesServedNextCycle
                        else R.ClassesServedCurrently
                    end
            N"R.AttachedProcess =
                    if DeviceReleased
                        then URProcess
                        else R.AttachedProcess
                    end

        KernelCalled(ReleaseDevice)
        if (~DeviceReleased)
            or
            R.State = Draining
            then KernelCalled[SendMessage(OpProcess)]
                    KernelCalled[SendMessage(R.AttachedProcess)]
        end
```

OP14a: Drain (Reader)

        given: Raddr: DeviceAddress

        entry: for some (R:ReaderEntry) in Readers:
                        R.Raddr = Raddr

        error on R.State = NotSpooling

        action: if R.CyclePosition = Available
                    then R.State <- Drained
                            KernelCall[SendMessage(
                                    Drained(R.Raddr),
                                    OpProcess)]
                    else R.State <- Draining
                 end

        exit: N"R.State =
                            if R.CyclePosition = Available
                                then Drained
                                else Draining
                            end

                 if R.CyclePosition = Available
                    then KernelCalled[SendMessage(OpProcess)]
                 end

OP15a: Start (Re der)

        given: Raddr: DeviceAddress
         .   NewClasses: set of Class

        entry: for some (R:ReaderEntry) in Readers:
                    R.Raddr = Raddr

        error on R.State = NotSpooling

        action: if Empty[NewClasses]
                then if Empty[ClassesServedCurrently]
                        then KernelCall[SendMessage(
                                        MustProvideInitialClassList[
                                              R.Raddr],
                                        OpProcess)]
                        else R.State <- Started
                                KernelCall[SendMessage(
                                        Started[R.Raddr],
                                        OpProcess)]
                    end
                else R.ClassesServedNextCycle <- NewClasses
                    if R.CyclePosition = Available
                        then R.ClassesServedCurrently <-
                                NewClasses
                    end
                    R.State <- Started
                    KernelCall[SendMessage(
                                Started[R.Raddr],
                                OpProcess)]
                end

68

```
exit:    N"R.State =
                if (~Empty[NewClasses])
                   or
                   (~Empty[R.ClassesServedCurrently])
                   then Started
                   else R.State
                end
         N"R.ClassesServedNextCycle =
                if ~Empty[NewClasses]
                   then NewClasses
                   else R.ClassesServedNextCycle
                end
         N"R.ClassesServedCurrently =
                if (~Empty[NewClasses])
                   &
                   R.CyclePosition = Available
                   then NewClasses
                   else R.ClassesServedCurrently
                end

KernelCalled[SendMessage(OpProcess)]
```

PRPU1: Process request for output spooling device assignment

        given: Process: ProcessName
                 RequestedClasses: set of Class

        entry: true

        error on for all (N:NkcpEntry) in CurrentNkcps:
                    (N.Process ~= Process)
                or
                Empty[RequestedClasses]

        action: if ~ ShuttingDown
                   then PrinterSpoolRequests <-
                                Append[PrinterSpoolRequests,Entry]
                end

        exit: N"PrinterSpoolRequests =
                        if ~ ShuttingDown
                           then Append[PrinterSpoolRequests,Entry]
                           else PrinterSpoolRequests
                        end

        where
        Entry = <Process = Process,
                 RequestedClasses = RequestedClasses,
                 AttachedDevice = ??,
                 State = WaitingForDevice>

70

PRPU2:   Printer Assignment (for spooling)

        given: DR: OutputDeviceRequest in PrinterSpoolRequests
               P: PrinterEntry in Printers

        entry: P.State = Started
                &
                P.CyclePosition = Available
                &
                DR.State = WaitingForDevice
                &
                ClassesMatch(P.ClassesServedCurrently,
                        DR.RequestedClasses)

        action: let (N:NkcpEntry) in CurrentNkcps:
                    (N.Process = DR.Process) in
                if for some (D:DeviceAddress) in N.UsablePrinters:
                        D = P.Raddr
                    then P.AttachedProcess <- DR.Process
                        P.CyclePosition <- SecurityHeader
                        DR.AttachedDevice <- P.Raddr
                        DR.State <- Processing
                        KernelCall(RequestIO(P.Raddr) for output)
                end

71

```
exit:   N"P.AttachedProcess =
                if for some  (D:DeviceAddress)
                                in N.UsablePrinters:
                    D = P.Raddr
                then DR.Process
                else P.AttachedProcess
            end
        N"P.CyclePosition =
                if for some  (D:DeviceAddress)
                                in N.UsablePrinters:
                    D = P.Raddr
                then SecurityHeader
                else P.CyclePosition
            end
        N"DR.AttachedDevice =
                if for some  (D:DeviceAddress)
                                in N.UsablePrinters:
                    D = P.Raddr
                then P.Raddr
                else DR.AttachedDevice
            end
        N"DR.State =
                if for some  (D:DeviceAddress)
                                in N.UsablePrinters:
                    D = P.Raddr
                then Processing
                else DR.State
            end

    KernelCall [RequestiO(P.Raddr) for output]
```

PRPU3a: Interrupt indicating end of security header output on
printer

        given: Raddr: DeviceAddress

        entry: for some (P:PrinterEntry) in Printers:
                    (P.Raddr = Raddr
                     &
                     P.State ~= NotSpooling
                     &
                     P.CyclePosition = SecurityHeader
                     &
                     P.ChannelStatusWord.UnitCheck = false)

        action: KernelCall(GrantAccess(P.AttachedProcess,P.Raddr))
                if OK
                    then P.CyclePosition <- AttachedToSpoolingProcess
                         KernelCall(SendMessage(
                                SpoolingDeviceAttached(P.Raddr),
                                P.AttachedProcess))
                         if P.RelinquishDeviceRequestState = ShouldSend
                            then P.RelinquishDeviceRequestState <- Sent
                                 KernelCall(SendMessage(
                                                Drain(P.Raddr),
                                                P.AttachedProcess))
                         end
                    else /* Kernel did not grant access as expected */
                         KernelCall(RequestIO(P.Raddr) for output)
                         P.CyclePosition <- SecurityTrailer
                         let (DR:OutputDeviceEntry)
                                        in PrinterSpoolRequests:
                             (DR.State = Processing
                              &
                              DR.AttachedDevice = P.Raddr
                              &
                              P.AttachedProcess = DR.Process) in
                         DR.State <- WaitingForDevice
                end

73

```
exit:    N"P.CyclePosition =
                if GrantedAccess
                   then AttachedToSpoolingProcess
                   else SecurityTrailer
                end
         N"P.RelinquishDeviceRequestState =
                if GrantedAccess
                   &
                   P.RelinquishDeviceRequestState = ShouldSend
                   then Sent
                   else P.RelinquishDeviceRequestState
                end

         KernelCalled[GrantAccess]
         if GrantedAccess
            then KernelCalled[SendMessage(P.AttachedProcess)]
                    if P.RelinquishDeviceRequestState = ShouldSend
                       then KernelCalled[SendMessage(
                                    P.AttachedProcess)]
                    end
            else KernelCalled[RequestIO(P.Raddr) for output]
         end
```

74

PRPU3b: Interrupt. IO error on attempt to output security header

given: Raddr: DeviceAddress

entry: for some (P:PrinterEntry) in Printers:
              (P.Raddr = Raddr
               &
               P.State ~= NotSpooling
               &
               P.CyclePosition inset (SecurityHeader,
                                        SecurityHeaderWaitForReady)
               &
               P.ChannelStatusWord.UnitCheck = true)

action: P.CyclePosition <- SecurityHeaderWaitForReady
          KernelCall(SendMessage(InterventionRequired[P.Raddr],
              OpProcess))

exit: N*P.CyclePosition = SecurityHeaderWaitForReady

          KernelCalled(SendMessage(OpProcess))

PRPU3c: Interrupt indicating OK to retry security header output

       given: Raddr: DeviceAddress

       entry: for some (P:PrinterEntry) in Printers:
                    (P.Raddr = Raddr
                     &
                     P.State ~= NotSpooling
                     &
                     P.CyclePosition = SecurityHeaderWaitForReady
                     &
                     P.ChannelStatusWord.UnitCheck = false)

       action: P.CyclePosition <- SecurityHeader
                KernelCall [RequestIO(P.Raddr) for output]

       exit: N"P.CyclePosition = SecurityHeader

                KernelCalled[RequestIO(P.Raddr) for output]

PRPU4: Process message, release output spooling device

given: Raddr: DeviceAddress
         Process: ProcessName

entry: true

error on ~(for some (P:PrinterEntry) in Printers:
              (P.Raddr = Raddr
               &
               P.State ~= NotSpooling
               &
               P.CyclePosition = AttachedToSpoolingProcess
               &
               P.AttachedProcess = Process)
           &
           for some (DR:OutputDeviceRequest)
                        in PrinterSpoolRequests:
              (DR.Process = P.AttachedProcess
               &
               DR.State = Processing
               &
               DR.AttachedDevice = P.Raddr))

action: P.CyclePosition <- SecurityTrailer
         P.AttachedProcess <- URProcess
         KernelCall [RequestIO(P.Raddr) for output]
         PrinterSpoolRequests <- Remove[PrinterSpoolRequests,DR]

exit: N"P.CyclePosition = SecurityTrailer
         N"P.AttachedProcess = URProcess
         N"PrinterSpoolRequests =
                  Remove[PrinterSpoolRequests,DR]

         KernelCalled[RequestIO(P.Raddr) for output]

PRPU5a: Interrupt indicating successful completion of security
trailer

        given: Raddr: DeviceAddress

        entry: for some (P:PrinterEntry) in Printers:
                    (P.Raddr = Raddr
                     &
                     P.State ~= NotSpooling
                     &
                     P.CyclePosition = SecurityTrailer
                     &
                     P.ChannelStatusWord.UnitCheck = false)

        action: if P.State = Draining
                    then P.State <- Drained
                end
                P.ClassesServedCurrently <- P.ClassesServedNextCycle
                P.RelinquishDeviceRequestState <- NoNeed
                P.CyclePosition <- Available

        exit: N"P.State =
                        if P.State = Draining
                            then Drained
                            else P.State
                        end
                N"P.ClassesServedCurrently = P.ClassesServedNextCycle
                N"P.RelinquishDeviceRequestState = NoNeed
                N"P.CyclePosition = Available

78

PRPU5b: Interrupt, IO error on security trailer output

given: Raddr: DeviceAddress

entry: for some (P:PrinterEntry) in Printers:
             (P.Raddr = Raddr
             &
             P.State ~= NotSpooling
             &
             P.CyclePosition inset (SecurityTrailer,
                                           SecurityTrailerWaitForReady)
             &
             P.ChannelStatusWord.UnitCheck = true)

action: P.CyclePosition <- SecurityTrailerWaitForReady
         KernelCall[SendMessage(
                 InterventionRequired[P.Raddr],
                 OpProcess)]

exit: N"P.CyclePosition = SecurityTrailerWaitForReady

         KernelCalled[SendMessage(OpProcess)]

PRPU5c: Interrupt, OK to retry security trailer output

       given: Raddr: DeviceAddress

       entry: for some (P:PrinterEntry) in Printers:
              (P.Raddr = Raddr
                                                      &
                P.State ~= NotSpooling
                                                    &
                P.CyclePosition = SecurityTrailerWaitForReady
                &
                P.ChannelStatusWord.UnitCheck = false)

      action: P.CyclePosition <- SecurityTrailer
               KernelCall[RequestIO(P.Raddr) for output]

       exit: N"P.CyclePosition = SecurityTrailer

              KernelCalled[RequestIO(P.Raddr) for output]

OP14bc: Drain (Printer, Punch)

        given: Raddr: DeviceAddress

        entry: for some (P:OutputDeviceEntry) in OutputDevices:
                    P.Raddr = Raddr

        error on P.State = NotSpooling

        action: if P.CyclePosition = Available
                    then P.State <- Drained
                        KernelCall (SendMessage(
                                Drained(P.Raddr),
                                OpProcess))
                    else if P.CyclePosition = AttachedToSpoolingProcess
                        then if P.RelinquishDeviceRequestState
                                ~= Sent
                                then P.RelinquishDeviceRequestState
                                        <- Sent
                                    KernelCall (SendMessage(
                                            Drain(P.Raddr),
                                            P.AttachedProcess))
                            end
                        else P.RelinquishDeviceRequestState
                                <- ShouldSend
                    end
                    P.State <- Draining
            end

```
exit:   N"P.State =
                if P.CyclePositicn = Available
                    then Draineo
                    else Draining
              end
        N"P.RelinquishDeviceRequestState =
                if P.CyclePosition =
                        AttachedToSpoolingProcess
                    then Sent
                    else if P.CyclePosition = Available
                            then P.RelinquishDeviceRequestState
                            else ShouldSend
                        end
              end

        if P.CyclePosition = Available
            then KernelCalled[SendMessage(OpProcess)]
        end
        if P.CyclePosition = AttachedToSpoolingProcess
            &
            P.Re''  ~uishDeviceRequestState ~= Sent
            the        :Called[SendMessage(P.AttachedProcess)]
        end
```

82

OP15bc: Start (Printer, Punch)

        given: Raddr: DeviceAddress
                 NewClasses: set of Class

        entry: for some (P:OutputDeviceEntry) in OutputDevices:
                     P.Raddr = Raddr

        error on P.State = NotSpooling

        action: if Empty[NewClasses]
                   &
                   Empty[P.ClassesServedCurrently]
                   then KernelCall[SendMessage(
                               MustProvideInitialClassList[P.Raddr],
                               OpProcess))
                   else P.State <- Started
                       KernelCall[SendMessage(
                                       Started[P.Raddr],
                                       OpProcess))
                         if ~Empty[NewClasses]
                            then P.ClassesServedNextCycle
                                  <- NewClasses
                                 if P.CyclePosition = Available
                                    then P.ClassesServedCurrently
                                          <- NewClasses
                                    else SendRequest[]
                                 end
                       end
                end

        macro SendRequest[] =
                if P.CyclePosition = AttachedToSpoolingProcess
                   then if P.RelinquishDeviceRequestState ~= Sent
                           then P.RelinquishDeviceRequestState <- Sent
                                KernelCall[SendMessage(
                                        Drain[P.Raddr],
                                        P.AttachedProcess)]
                        end
                   else P.RelinquishDeviceRequestState <- ShouldSend
                end

```
exit:   N"P.State =
                if (~Empty(NewClasses))
                or
                (~Empty(P.ClassesServedCurrently))
                then Started
                else P.State
            end
        N"P.ClassesServedNextCycle =
                if ~Empty(NewClasses)
                    then NewClasses
                    else P.ClassesServedNextCycle
            end
        N"P.ClassesServedCurrently =
                if P.CyclePosition = Available
                    &
                    ~Empty(NewClasses)
                    then NewClasses
                    else P.ClassesServedCurrently
            end
        N"P.RelinquishDeviceRequestState =
                if ~Empty(NewClasses)
                    then if P.CyclePosition =
                                    AttachedToSpoolingProcess
                            then Sent
                            else if P.CyclePosition =
                                            Available
                                    then
                                P.RelinquishDeviceRequestState
                                        else ShouldSend
                                    end
                        end
                    else P.RelinquishDeviceRequestState
            end

    KernelCalled(SendMessage(OpProcess))
    if (~Empty(NewClasses))
        &
        P.CyclePosition = AttachedToSpoolingProcess
        &
        P.RelinquishDeviceRequestState ~= Sent
        then KernelCalled(SendMessage(P.AttachedProcess))
    end
```

84

OP0:   Miscellaneous commands

        commands:
                QUERY←UR
                QUERY←ALL
                QUERY←RADDR
                QUERY←TAPES

        entry: true

        exit: KernelCalled(SendMessage(OpProcess))

85

OP1:   Single message sent
                    No maps
                    Single response expected
                    No device state information modifications

          commands:
                    QUERY←READER←SPOOLID
                    QUERY←PRINTER←SPOOLID
                    QUERY←PUNCH←SPOOLID
                    CHANGE←SYSTEM←SPOOLID

          entry: true

          exit: N"PendingRequests = Append[PendingRequests,
                              <MsgId = new[MessageId],
                               Kind = OpRequest,
                               Command = Command,
                               Responses = [
                                       <Respondent = Destination[Command],
                                        Text = nil,
                                        State = NoResponse>] >]

          KernelCalled[SendMessage[Destination[Command]]]

OP2:   Multiple messages sent
                No maps
                Responses expected
                No device state information modifications

        commands:
                QUERY←FILES←ALL
                QUERY←READER←ALL
                QUERY←PRINTER←ALL
                QUERY←PUNCH←ALL
                QUERY←HOLD
                CHANGE←SYSTEM←CLASS←ALL

        entry: true

        exit: N"PendingRequests = Append[PendingRequests,
                        <MsgId = new[MessageId],
                         Kind = OpRequest,
                         Command = Command,
                         Responses = [for all (N:ProcessName)
                                                  in CurrentNkcps:
                                <Respondent = N,
                                 Text = nil,
                                 State = NoResponse>] >]

                for all (N:ProcessName) in CurrentNkcps:
                    KernelCalled[SendMessage(N)]

87

OP3:   Single message sent
                  Mapping: User Id -> Nkcp Id
                  Single response expected
                  No device state information modifications

     commands:
                  QUERY←FILES←USERID
                  QUERY←READER←USERID
                  QUERY←PRINTER←USERID
                  QUERY←PUNCH←USERID
                  CHANGE←USERID
                  FREE
                  HOLD
                  ORDER←USERID
                  PURGE←USERID

     entry: true

     exit: N"PendingRequests = Append(PendingRequests,
                        <MsgId = new(MessageId),
                         Kind = MapUserId,
                         Command = Command,
                         Responses = {
                                <Respondent = AuthProcess,
                                 Text = nil,
                                 State = NoResponse>}>]

     KernelCalled(SendMessage(AuthProcess)]

OP4ab: commands:
                BACKSPAC
                FLUSH
                REPEAT

    given: Raddr: DeviceAddress

    entry: for some (P:OutputDeviceEntry) in OutputDevices:
                P.Raddr = Raddr

    error on ~(P.State ~= NotSpooling
                &
                P.CyclePosition = AttachedToSpoolingProcess)

    exit: N"PendingRequests = Append(PendingRequests,
                    <MsgId = new(MessageId),
                    Kind = OpRequest,
                    Command = Command,
                    Responses = {
                            <Respondent = P.AttachedProcess,
                            Text = nil,
                            State = NoResponse>} >)

        KernelCalled(SendMessage(P.AttachedProcess))

OP5:   command:
                SPACE

        given: Raddr: DeviceAddress

        entry: for some (P:PrinterEntry) in Printers:
                    P.Raddr = Raddr

        error on ~(P.State ~= NotSpooling
                        &
                    P.CyclePosition = AttachedToSpoolingProcess)

        exit: N"PendingRequests = Append(PendingRequests,
                        <MsgId = new(MessageId),
                         Kind = OpRequest,
                         Command = SPACE,
                         Responses = (
                                <Respondent = P.AttachedProcess,
                                 Text = nil,
                                 State = NoResponse>) >)

            KernelCalled(SendMessage(P.AttachedProcess))

OP6:   commands:
              ORDER←SYSTEM
              PURGE←SYSTEM

      given: Nkcps: set of ProcessName

      entry: true

      exit: N"PendingRequests = Append(PendingRequests,.
                        <MsgId = new(MessageId),
                        Kind = OpRequest,
                        Command = Command,
                        Responses = {for all (N:ProcessName) in Nkcps:
                                  <Respondent = N,
                                    Text = nil,
                                  State = NoResponse>} >}

            for all (N:ProcessName) in Nkcps:
              KernelCalled{SendMessage(N)}

91

OP7abc: LOCATE of Reader, Printer, Punch

        given: Raddr: DeviceAddress

        entry: for some (D:DeviceEntry) in Devices:
                        D.Raddr = Raddr

        error on D.CyclePosition ~inset {AttachedToSpoolingProcess,
                                         AttachedToUser}

        exit: N"PendingRequests = Append(PendingRequests,
                        <MsgId = new(MessageId),
                         Kind = OpRequest,
                         Command = LOCATE,
                         Responses = {
                                <Respondent = D.AttachedProcess,
                                 Text = nil,
                                 State = NoResponse>}>}

        KernelCalled(SendMessage(D.AttachedProcess))

92

OP7d: LOCATE of Tape Drive

given: Raddr: DeviceAddress

entry: for some (T:TapeDriveEntry) in TapeDrives:
                T.Raddr = Raddr

error on T.State ~= AttachedToUser

exit: N"PendingRequests = Append[PendingRequests,
                <Msgld = new[Messageld],
                 Kind = OpRequest,
                 Command = LOCATE,
                 Responses = {
                        <Respondent = T.AttachedProcess,
                         Text = nil,
                         State = NoResponse>}>]

        KernelCalled[SendMessage(T.AttachedProcess)]

OP8:   command:
                    SHUTDOWN

          entry: true

          action: ShuttingDown <- true
                    for all (DR:OutputDeviceRequest)
                                            in PrinterSpoolRequests:
                        if DR.State = WaitingForDevice
                           then PrinterSpoolRequests <- Remove[
                                            PrinterSpoolRequests,DR]
                    end
                    for all (DR:OutputDeviceRequest)
                                            in PunchSpoolRequests:
                        if DR.State = WaitingForDevice
                           then PunchSpoolRequests <- Remove[
                                            PunchSpoolRequests,DR]
                    end

          exit: N"ShuttingDown = true

                    for all (DR:OutputDeviceEntry) in PrinterSpoolRequests:
                        N"DR.State ~= WaitingForDevice
                    for all (DR:OutputDeviceEntry) in PunchSpoolRequests:
                        N"DR.State ~= WaitingForDevice

OP9abc: Vary offline (Reader, Printer, Punch)

        given: Raddr: DeviceAddress

        entry: for some (D:DeviceEntry) in Devices:
                        D.Raddr = Raddr

        error on D.CyclePosition ~inset {Available,OffLine}

        action: D.CyclePosition <- OffLine
                 D.State <- NotSpooling
                 KernelCall(SendMessage(
                         OffLine(D.Raddr),
                         OpProcess))

        exit: N"D.CyclePosition = OffLine
                 N"D.State = NotSpooling

                 KernelCalled(SendMessage(OpProcess))

OP9d: Vary offline (Tape Drive)

        given: Raddr: DeviceAddress

        entry: for some (T:TapeDriveEntry) in TapeDrives:
                        T.Raddr = Raddr

        error on T.State ~inset (Available,OffLine)

        action: T.State <- OffLine
                 KernelCall [SendMessage(
                        OffLine(T.Raddr),
                        OpProcess)]

        exit: N"T.State = OffLine

                 KernelCalled(SendMessage(OpProcess))

OP10a: Vary online (Reader)

given: Raddr: DeviceAddress

entry: for some (R:ReaderEntry) in Readers:
                R.Raddr = Raddr

error on ~(R.State = NotSpooling
                &
                R.CyclePosition = OffLine)

action: R.State <- Drained
         R.CyclePosition <- Available
         R.ClassesServedCurrently <- R.ClassesServedNextCycle
         R.AttachedProcess <- URProcess
         KernelCall[SendMessage(OnLine[R.Raddr],OpProcess)]

exit: N"R.State = Drained
         N"R.CyclePosition = Available
         N"R.ClassesServedCurrently = R.ClassesServedNextCycle
         N"R.AttachedProcess = URProcess

         KernelCalled[SendMessage(OpProcess)]

97

OP10bc: Vary online (Output Device)

        given: Raddr: DeviceAddress

        entry: for some (D:OutputDeviceEntry) in OutputDevices:
                    D.Raddr = Raddr

        error on ~(D.State = NotSpooling
                        &
                    D.CyclePosition = OffLine)

        action: D.State <- Drained
                D.CyclePosition <- Available
                D.ClassesServedCurrently <- D.ClassesServedNextCycle
                D.AttachedProcess <- URProcess
                D.RelinquishDeviceRequestState <- NoNeed
                KernelCall[SendMessage(OnLine[D.Raddr],OpProcess)]

        exit: N"D.State = Drained
                N"D.CyclePosition = Available
                N"D.ClassesServedCurrently = D.ClassesServedNextCycle
                N"D.AttachedProcess = URProcess
                N"D.RelinquishDeviceRequestState = NoNeed

                KernelCalled(SendMessage(OpProcess)]

OP10d: Vary online (Tape Drive)

        given: Raddr: DeviceAddress

        entry: for some (T:TapeDriveEntry) in TapeDrives:
                     (T.Raddr = Raddr)

        error on ~(T.State = OffLine)

        action: T.State <- Available
                 T.AttachedProcess <- URProcess
                 KernelCall(SendMessage(
                         OnLine(T.Raddr),
                         OpProcess))

        exit: N"T.State = Available
                N"T.AttachedProcess = URProcess

                KernelCalled(SendMessage(OpProcess))

99

AUTH3abc: Attach device   to process (request from AuthProcess)
(Reader, Printer, Punch)

        given: Raddr: DeviceAddress
                 Process: ProcessName

        entry: for some (D:DeviceEntry) in Devices:
                    D.Raddr = Raddr

        error on for all (N:NkcpEntry) in CurrentNkcps:
                    N.Process ~= Process

        action: if D.State = Drained
                    &
                    D.CyclePosition = Available
                    then let (N:NkcpEntry) in CurrentNkcps:
                            (N.Process = Process) in
                        KernelCall (GrantAccess(D.Raddr.N.Process)]
                        if OK
                            then D.State <- NotSpooling
                                 D.CyclePosition <- AttachedToUser
                                 D.AttachedProcess <- N.Process
                                 KernelCall (SendMessage(
                                         Attached(D.Raddr,
                                                     N.Process],
                                         AuthProcess)]
                            else /* Kernel did not allow the access */
                                 KernelCall (SendMessage(
                                         AttachFailed(D.Raddr,N.Process],
                                         AuthProcess)]
                        end
                else /* device not available at this time */
                    KernelCall (SendMessage(
                                 DeviceNotAvailable(D.Raddr],
                                 AuthProcess)]
            end

```
exit:    N"D.State =
                    if D.State = Drained
                        &
                        D.CyclePosition = Available
                        &
                        GrantedAccess
                        then NotSpooling
                        else D.State
                    end
          N"D.CyclePosition =
                    if D.State = Drained
                        &
                        D.CyclePosition = Available
                        &
                        GrantedAccess
                        then AttachedToUser
                        else D.CyclePosition
                    end
          N"D.AttachedProcess =
                    if D.State = Drained
                        &
                        D.CyclePosition = Available
                        &
                        GrantedAccess
                        then N.Process
                        else D.AttachedProcess
                    end

        if D.State = Drained
            &
            D.CyclePosition = Available
            then KernelCalled[GrantAccess(D.Raddr)]
        end
        KernelCalled[SendMessage(AuthProcess)]
```

AUTH3d: Attach tape drive (request from AuthProcess)

        given: Raddr: DeviceAddress
                Process: ProcessName
                ReqAccess: AccessModes

        entry: for some (T:TapeDriveEntry) in TapeDrives:
                    T.Raddr = Raddr

        error on for all (N:NkcpEntry) in CurrentNkcps:
                    N.Process ~= Process

        action: let (N:NkcpEntry) in CurrentNkcps:
                    (N.Process = Process) in
                if T.State = Available
                    then KernelCall[GrantAccess(
                                T.Raddr,
                                N.Process,
                                ReqAccess)]
                        if OK
                            then T.State <- AttachedToUser
                                T.AttachedProcess <- N.Process
                                KernelCall[SendMessage(
                                        Attached[T.Raddr],
                                        AuthProcess)]
                            else KernelCall[SendMessage(
                                        NotOK[T.Raddr],
                                        AuthProcess)]
                        end
                    else KernelCall[SendMessage(
                                DriveNotAvailable[T.Raddr],
                                AuthProcess)]
                end

```
exit:    N"T.State =
                  if T.State = Available
                     &
                     GrantedAccess
                     then AttachedToUser
                     else T.State
                  end
         N"T.AttachedProcess =
                  if T.State = Available
                     &
                     GrantedAccess
                     then N.Process
                     else T.AttachedProcess
                  end

         KernelCalled[SendMessage(AuthProcess)]
```

OP11abc: Attach device to process (request from   operator)
(Reader, Printer, Punch)

        given: Raddr: DeviceAddress
                Process: ProcessName

        entry: for some (D:DeviceEntry) in Devices:
                    D.Raddr = Raddr

        error on for all (N:NkcpEntry) in CurrentNkcps:
                    N.Process ~= Process

        action: let (N:NkcpEntry) in CurrentNkcps:
                    (N.Process = Process)
                 if D.State ~ Drained
                    &
                    D.CyclePosition = Available
                    then if for some (A:DeviceAddress)
                            in union of (N.UsableReaders,
                                         N.UsablePrinters,
                                         N.UsablePunches):
                            (A = D.Raddr)
                            then KernelCall [GrantAccess(
                                        D.Raddr,
                                        N.Process)]
                            if OK
                                then D.State <- NotSpooling
                                     D.CyclePosition
                                             <- AttachedToUser
                                     D.AttachedProcess
                                             <- N.Process
                                     KernelCall [SendMessage(
                                        Attached[
                                             D.Raddr,
                                             N.Process],
                                        OpProcess)]
                                     KernelCall [SendMessage(
                                        Attached[D.Raddr],
                                        N.Process)]
                                else KernelCall [SendMessage(
                                        AttachFailed[
                                             D.Raddr,
                                             N.Process],
                                        OpProcess)]
                            end
                      else KernelCall [SendMessage(
                                    DeviceNotUsable[
                                        D.Raddr,
                                        N.Process],
                                    OpProcess)]
                 end

                104

```
                    else KernelCall(SendMessage(
                              DeviceNotAvailable(D.Raddr),
                              OpProcess))
            end

  exit:    N"D.State =
                    if D.State = Drained
                         &
                         D.CyclePosition = Available
                         &
                         GrantedAccess
                         then NotSpooling
                         else D.State
                    end
            N"D.CyclePosition =
                    if D.State = Drained
                         &
                         D.CyclePosition = Available
                         &
                         GrantedAccess
                         then AttachedToUser
                         else D.CyclePosition
                    end
            N"D.AttachedProcess =
                    if D.State = Drained
                         &
                         D.CyclePosition = Available
                         &
                         GrantedAccess
                         then N.Process
                         else D.AttachedProcess
                    end

    KernelCalled(SendMessage(OpProcess))
    if D.State = Drained
         &
       D.CyclePosition = Available
       then KernelCalled(GrantAccess)
            if GrantedAccess
               then KernelCalled(SendMessage(N.Process))
            end
    end
```

OP12abc: Detach dedicated device from user   (request   from
operator) (Reader, Printer, Punch)

        given: Raddr: DeviceAddress
                 Process: ProcessName

        entry: for some (D:DeviceEntry) in Devices:
                     (D.Raddr = Raddr)

        error on ~(D.State = NotSpooling
                     &
                     D.CyclePosition = AttachedToUser
                     &
                     D.AttachedProcess = Process)

        action: D.CyclePosition <- DetachPending
                 KernelCall(SendMessage(
                         RelinquishDevice(D.Raddr),
                         D.AttachedProcess))

        exit: N"D.CyclePosition = DetachPending
                 N"PendingRequests = Append(PendingRequests,
                     <MsgId = new(MessageId),
                      Kind = RelinquishDevice,
                      Command = ??,
                      Responses = (
                             <Respondent = D.AttachedProcess,
                              Text = nil,
                              State = NoResponse>)>)

NKCP2abc: Detach dedicated device from user (request   from
attached process) (Reader, Printer, Punch)


```
         given:  Raddr: DeviceAddress
                 Process: ProcessName

         entry:  for some (D:DeviceEntry) in Devices:
                     (D.Raddr = Raddr)

         error on ~(D.State = NotSpooling
                       &
                    D.CyclePosition = AttachedToUser
                       &
                    D.AttachedProcess = Process)

         action: KernelCall(ReleaseDevice(
                         D.Raddr,
                         D.AttachedProcess))
                 if OK
                    then D.State <- Drained
                         D.CyclePosition <- Available
                         D.AttachedProcess <- URProcess
                         KernelCall(SendMessage(
                                 Detached(D.Raddr),
                                 D.AttachedProcess))
                         KernelCall(SendMessage(
                                 Detached(D.Raddr),
                                 OpProcess))
                    else KernelCall(SendMessage(
                                 DeviceNotReleasable(D.Raddr),
                                 D.AttachedProcess))
                         KernelCall(SendMessage(
                                 DeviceNotReleasable[
                                         D.Raddr,
                                         D.AttachedProcess],
                                 OpProcess))
                 end
```

```
exit:    N"D.State =
                if DeviceReleased
                    then Drained
                    else D.State
                end
         N"D.CyclePosition =
                if DeviceReleased
                    then Available
                    else D.CyclePosition
                end
         N"D.AttachedProcess =
                if DeviceReleased
                    then URProcess
                    else D.AttachedProcess
                end

         KernelCalled(ReleaseDevice)
         KernelCalled(SendMessage(D.AttachedProcess))
         KernelCalled(SendMessage(OpProcess))
```

OP12d: Detach dedicated device from user (request from operator)
(Tape Drive)

```
        given:  Raddr: DeviceAddress
                Process: ProcessName

        entry:  for some (T:TapeDriveEntry) in TapeDrives:
                    (T.Raddr = Raddr)

        error on ~(T.State = Attached
                        &
                    T.AttachedProcess = Process)

        action: T.State <- DetachPending
                KernelCall(SendMessage(
                        RelinquishDevice(T.Raddr),
                        T.AttachedProcess))

        exit:   N"T.State = DetachPending
                N"PendingRequests = Append(PendingRequests,
                        <MsgId = new(MessageId),
                         Kind = RelinquishDevice,
                         Command = ??,
                         Responses = {
                                <Respondent = T.AttachedProcess,
                                 Text = nil,
                                 State = NoResponse>}>}

            KernelCalled(SendMessage(T.AttachedProcess))
```

109

NKCP2d: Detach dedicated device from user (request from   attached
process) (Tape Drive)

given: Raddr: DeviceAddress
         Process: ProcessName

entry: for some (T:TapeDriveEntry) in TapeDrives:
              (T.Raddr = Raddr)

error on ~(T.State = Attached
              &
            T.AttachedProcess = Process)

action: KernelCall(ReleaseDevice(
                    T.Raddr,
                    T.AttachedProcess))
         if OK
            then T.State <- Available
                 T.AttachedProcess <- URProcess
                 KernelCall(SendMessage(
                         Detached(T.Raddr),
                         T.AttachedProcess))
                 KernelCall(SendMessage(
                         Detached(T.Raddr),
                         OpProcess))
            else KernelCall(SendMessage(
                         DeviceNotReleasable(T.Raddr),
                         T.AttachedProcess))
                 KernelCall(SendMessage(
                         DeviceNotReleasable
                                 T.Raddr,
                                 T.AttachedProcess),
                         OpProcess))
         end

exit: N"T.State =
                 if DeviceReleased
                    then Available
                    else T.State
                 end
         N"T.AttachedProcess =
                 if DeviceReleased
                    then URProcess
                    else T.AttachedProcess
                 end

         KernelCalled(ReleaseDevice)
         KernelCalled(SendMessage(T.AttachedProcess))
         KernelCalled(SendMessage(OpProcess))

                              118

NKCP1abc: Process message, relinquishing device as requested
(Reader, Printer, Punch)

```
given:   Raddr: DeviceAddress
         Process: ProcessName

entry:   for some (D:DeviceEntry) in Devices:
             D.Raddr = Raddr

error on ~(D.State = NotSpooling
              &
           D.CyclePosition = DetachPending
              &
           D.AttachedProcess = Process)

action: KernelCall [ReleaseDevice(D.Raddr,D.AttachedProcess)]
        if OK
            then KernelCall [SendMessage(
                        Detached[D.Raddr],
                        OpProcess)]
                 KernelCall [SendMessage(
                        Detached[D.Raddr],
                        D.AttachedProcess)]
                 D.State <- Drained
                 D.CyclePosition <- Available
                 D.AttachedProcess <- URProcess
            else KernelCall [SendMessage(
                        DeviceNotReleasable[D.Raddr],
                        D.AttachedProcess)]
                 KernelCall [SendMessage(
                        DeviceNotReleasable[
                            D.Raddr,
                            D.AttachedProcess],
                        OpProcess)]
        end
```

111

```
exit:    N"D.State =
                    if DeviceReleased
                        then Drained
                        else D.State
                    end
         N"D.CyclePosition =
                    if DeviceReleased
                        then Available
                        else D.CyclePosition
                    end
         N"D.AttachedProcess =
                    if DeviceReleased
                        then URProcess
                        else D.AttachedProcess
                    end

         KernelCalled(ReleaseDevice)
         KernelCalled(SendMessage(D.AttachedProcess))
         KernelCalled(SendMessage(OpProcess))
```

112

CP13: Loadbuf

given: Raddr: DeviceAddress

entry: for some (P:PrinterEntry) in Printers:
P.Raddr = Raddr

error on ~(P.State = Drained
&
P.CyclePosition = Available)

exit: true

NKCP1d: Process message, relinquishing device as requested (Tape
Drive)

        given: Raddr: DeviceAddress
                 Process: ProcessName

        entry:. for some (T:TapeDriveEntry) in TapeDrives:
                    T.Raddr = Raddr

        error on ~(T.State = DetachPending
                     &
                   T.AttachedProcess = Process)

        action: KernelCall(ReleaseDevice(T.Raddr,Process))
                if OK
                    then KernelCall(SendMessage(
                                 Detached(T.Raddr),
                                 OpProcess))
                         KernelCall(SendMessage(
                                 Detached(T.Raddr),
                                 T.AttachedProcess))
                         T.State <- Available
                         T.AttachedProcess <- URProcess
                    else KernelCall(SendMessage(
                                 DeviceNotReleasable(T.Raddr)
                                 T.AttachedProcess))
                         KernelCall(SendMessage(
                                 DeviceNotReleasable(
                                         T.Raddr,
                                         T.AttachedProcess),
                                 OpProcess))
                end

        exit: N"T.State =
                         if DeviceReleased
                            then Available
                            else T.State
                         end
                 N"T.AttachedProcess =
                         if DeviceReleased
                            then URProcess
                            else T.AttachedProcess
                         end

        KernelCalled(ReleaseDevice)
        KernelCalled(SendMessage(T.AttachedProcess))
        KernelCalled(SendMessage(OpProcess))

                              114

OP11d: Attach tape drive (request from operator)

        given: Raddr: DeviceAddress
                 ReqAccess: AccessModes
                 TapeSecLevel: ProcessName
                 Process: ProcessName

        entry: for some (T:TapeDriveEntry) in TapeDrives:
                     T.Raddr = Raddr

        error on for all (N:NkcpEntry) in CurrentNkcps:
                     N.Process ~= Process

        action: let (N:NkcpEntry) in CurrentNkcps:
                     (N.Process = Process) in
                 if for some (D:DeviceAddress) in N.UsableTapeDrives:
                        D = T.Raddr
                    then if T.State = Available
                            then KernelCall[CheckSecLevel(
                                        TapeSecLevel,
                                        ReqAccess,
                                        T.Raddr,
                                        N.Process)]
                                 if OK
                                    then KernelCall[GrantAccess(
                                                T.Raddr,
                                                N.Process,
                                                ReqAccess)]
                                         if OK
                                            then KernelCall[SendMessage(
                                                    Attached[T.Raddr],
                                                    N.Process)]
                                                 KernelCall[SendMessage(
                                                    Attached[
                                                        T.Raddr,
                                                        N.Process],
                                                    OpProcess)]
                                                 T.State <- Attached
                                                 T.AttachedProcess <-
                                                    N.Process
                                            else CannotAttach[NotOK]
                                         end
                                    else CannotAttach[
                                                CannotUseTape]
                                 end
                            else CannotAttach[TapeDriveNotAvailable]
                         end
                    else CannotAttach[CannotUseTapeDrive]
                 end

                                115

```
macro CannotAttach(Reason) =
        KernelCall(SendMessage(
                NotAttached(
                        T.Raddr,
                        N.Process,
                        Reason),
                OpProcess))

exit:   N"T.State =
                if for some (D:DeviceAddress)
                                in N.UsableTapeDrives:
                        (D = T.Raddr)
                  &
                  T.State = Available
                  &
                  CheckedSecLevel
                  &
                  GrantedAccess
                  then Attached
                  else T.State
                end
        N"T.AttachedProcess =
                if for some (D:DeviceAddress)
                                in N.UsableTapeDrives:
                        (D = T.Raddr)
                  &
                  T.State = Available
                  &
                  CheckedSecLevel
                  &
                  GrantedAccess
                  then Process
                  else T.AttachedProcess

KernelCalled(SendMessage(OpProcess))
if for some (D:DeviceAddress) in N.UsableTapeDrives:
        (D = T.Raddr)
   &
   T.State = Available
   then KernelCalled(CheckSecLevel)
        if CheckedSecLevel
            then KernelCalled(GrantAccess(T.Raddr))
                    if GrantedAccess
                        then KernelCalled(
                                SendMessage(Process))
                    end
            end
   end
```

116

AUTH1:   Add Nkcp

       given: Process: ProcessName
                Readers: set of DeviceAddress
                Printers: set of DeviceAddress
                Punches: set of DeviceAddress
                TapeDrives: set of DeviceAddress

       entry: true

       error on for some (N:NkcpEntry) in CurrentNkcps:
                     N.Process = Process

       exit: N"CurrentNkcps = Append(CurrentNkcps,
                     <Process = Process,
                      UsableReaders = Readers,
                      UsablePrinters = Printers,
                      UsablePunches = Punches,
                      UsableTapeDrives = TapeDrives>]

117

AUTH2:   Delete Nkcp

        given: Process: ProcessName

        entry: true

        error on for all (N:NkcpEntry) in CurrentNkcps:
                    N.Process ~= Process

        let (N:NkcpEntry) in CurrentNkcps:
            (N.Process = Process) in

        exit: N"CurrentNkcps = Remove(CurrentNkcps,N)
                 for all (OR:OutputDeviceEntry) in PrinterSpoolRequests:
                     N"OR.AttachedProcess ~= N.Process
                 for all (OR:OutputDeviceEntry) in PunchSpoolRequests:
                     N"OR.AttachedProcess ~= N.Process

118

KERN1: message from Kernel, re device availability (during scan
at system initialization) .

         given: Raddr: DeviceAddress

         entry: true

         action: if for some (D:DeviceEntry) in Devices:
                      D.Raddr = Raddr
                   then error on D.State ~= Drained
                      D.State <- NotAvailableForSpooling
                      D.CyclePosition <- OffLine
                   else if for some (T:TapeDriveEntry) in TapeDrives:
                             T.Raddr = Raddr
                          then error on T.State ~= Available
                                 T.State <- OffLine
                          else error
                      end
              end

         exit: if for some (D:DeviceEntry) in Devices:
                      (D.Raddr = Raddr)
                   then N"D.State = NotAvailableForSpooling
                      N"D.CyclePosition = OffLine
                   else if for some (T:TapeDriveEntry) in TapeDrives:
                             (T.Raddr = Raddr)
                          then N"T.State = OffLine
                      end
              end

                                    119

Authorization Process
Semi-Formal Description

This section contains a semi-formal description of the
Authorization Process of KVM/370.

Data Types

primitive types and structuring mechanisms:

        boolean   (unordered, two elements: true, false)
        string   (unbounded, predefined string of length zero: nil)
        integer   subrange

        scalar   (ordered element list)
        set   (of any type, predefined empty set: nil)
        record   (field list)

undefined types:

        DeviceAddress
        LineAddress
        ProcessName
        VirtualMachineName
        VolumeId

undefined functions / macros:

        Dominates
        DeviceType
        #Cylinders

```
CommandName: scalar(
        AUTOLOG,
        ATTACH←RADDR,
        DETACH←RADDR,
        VARY,
        QUERY←DASD,
        QUERY←LINES,
        QUERY←GRAF,
        QUERY←NAMES,
        QUERY←USERS←X,
        QUERY←ALL,
        QUERY←SYSTEM←RADDR,
        QUERY←RADDR,
        QUERY←USERS←USERID,
        QUERY←USERID,
        LOCATE←RADDR,
        SHUTDOWN)


RequestCategory: scalar(
        Attach,
        ClearLine,
        ReDirectLine,
        WriteAndReadLine,
        OpRequest,
        NewVM,
        ConnectVM,
        NewUser,
        NewOrConnectedVM,
        RelinquishDevice)


ResponseStatus: scalar(
        NoResponse,
        Responded)


AccessModes: scalar(
        Read,
        Write)
```

121

```
LineStatus: scalar(
        Retry,
        Disabled,
        Available,
        ReadInitialPassword,
        ReadAccessPassword,
        PerformResourceChecks,
        HookingPeripherals,
        NotifyingNkcp,
        Attached,
        ReadLinkPassword,
        ReEnablePending)


SharableDriveStatus: scalar(
        OffLine,
        Available,
        AttachedToSystem)


DriveStatus: scalar(
        OffLine,
        DetachPending,
        AttachedToUser,
        Available)


VolumeStatus: scalar(
        Mounted,
        NotMounted)


LinkAccess: scalar(R,RR,W,WR,M,MR,MW)

LineCondition: subset of LineStatus:(
        Disabled,
        Available)


ActivityStatus: scalar(
        Free,
        Attached,
        AttachValidation)


AccessCategory: scalar(
        Logon,
        Dial)
```

```
ReasonTypes: scalar(
        IncorrectLogon,
        ResourceFailure,
        SecurityViolation,
        MaxThresholdExceeded,
        NoNkcp,
        NoVM,
        TerminalClearanceMismatch)


LogoffReasons: scalar(
        UserChoice,
        Forced,
        Disconnected)


DirectoryEntry:
        record
                UserId: VirtualMachineName
                LogonPassword: string
                DialPassword: string
                LinkPassword: string
                MaxSecLevel: ProcessName
                MinSecLevel: ProcessName
                DedicatedDevices: set of DedicatedDeviceEntry
                Links: set of MDLinkEntry
                IplDefined:  oolean
                AccessPasswords: set of AccessPasswordEntry
        end

LineEntry:
        record
                Laddr: LineAddress
                MaxSecLevel: ProcessName
                MinSecLevel: ProcessName
                State: ActivityS'atus
                CyclePosition: LineStatus
                RequestedSecLevel: ProcessName
                AttachedVM: VirtualMachineName
                Connection: AccessCategory
                LineDropped: boolean
                #Retries: 0..#MaxRetries
                #AwaitingHooks: nonnegative integer
                Msg: string
        end
```

123

```
NkcpEntry:
        record
                Process: ProcessName
                VMs: set of VMEntry
                AttachedDevices: set of AttachedDeviceEntry
                Links: set of MDLinkEntry
        end


AccessPasswordEntry:
        record
                SecLevel: ProcessName
                Password: string
        end


VMEntry:
        record
                VMName: VirtualMachineName
                Laddr: LineAddress
                Disconnected: boolean
                Users: set of LineAddress
        end


DedicatedDeviceEntry:
        record
                Raddr: DeviceAddress
                VolSecLevel: ProcessName
                Access: set of AccessModes
        end


AttachedDeviceEntry:
        record
                Raddr: DeviceAddress
                Access: set of AccessModes
        end


MDLinkEntry:
        record
                MDName: MiniDiskName
                Access: set of AccessModes
        end


ProcessLinkEntry:
        record
                Process: ProcessName
                Access: set of AccessModes
        end
```

124

```
URPOwnedDeviceEntry:
        record
                Raddr: DeviceAddress
                MaxSecLevel: ProcessName
                MinSecLevel: ProcessName
        end


NonsharableDriveEntry:
        record
                Raddr: DeviceAddress
                MaxSecLevel: ProcessName
                MinSecLevel: ProcessName
                State: DriveStatus
                AttachedProcess: ProcessName
                Access: set of AccessModes
        end


SharableDriveEntry:
        record
                Raddr: DeviceAddress
                State: SharableDriveStatus
                SecLevel: ProcessName
                MountedVolume: VolumeId
        end


SharedVolumeEntry:
        record
                Volume: VolumeId
                SecLevel: ProcessName
                MountedDevice: DeviceAddress
                State: VolumeStatus
        end


MiniDiskEntry:
        record
                MDName: MiniDiskName
                ContainingVolume: VolumeId
                Cylinders: (1..#MaxCylinders,
                            1..#MaxCylinders)
                SecLevel: ProcessName
                CurrentLinks: set of ProcessLinkEntry
                AccessControlList: set of ACLEntry
        end
```

125

```
ACLEntry:
        record
                User: VirtualMachineName
                Access: set of AccessModes
        end


ResponseSlot:
        record
                Respondent: ProcessName
                Text: string
                State: ResponseStatus
        end


PendingRequest:
        record
                MsgId: MessageId
                Kind: RequestCategory
                Command: CommandName
                Responses: set of ResponseSlot
        end
```

Data Structures

constant AddressSpaceSize: 0..8192

constant CodeSize: 0..8192

constant #MaxCylinders: positive integer

constant Code: integer

constant #MaxRetries: nonnegative integer

constant #MaxNkcps: nonnegative integer

constant #MaxVMs: nonnegative integer

#Nkcps: 0..#MaxNkcps

#VMs: 0..#MaxVMs

#Users: nonnegative integer

ShuttingDown: boolean

URPOwnedDevices: set of URPOwnedDeviceEntry

NonsharableDrives: set of NonsharableDriveEntry

SharableDrives: set of SharableDriveEntry

SharedVolumes: set of SharedVolumeEntry

MiniDisks: set of MiniDiskEntry

CurrentNkcps: set of NkcpEntry

Lines: set of LineEntry

UserDirectory: set of DirectoryEntry

PendingRequests: set of PendingRequest

Initial Conditions

```
#Nkcps = 0
&
#VMs = 0
&
#Users = 0
&
(~ShuttingDown)
&
for all (NS:NonsharableDriveEntry) in NonsharableDrives:
    (NS.State = Available
     &
     NS.AttachedProcess = AuthProcess)
&
for all (S:SharableDriveEntry) in SharableDrives:
    (S.State = Available)
&
for all (V:SharedVolumeEntry) in SharedVolumes:
    (V.State = NotMounted)
&
for all (M:MiniDiskEntry) in MiniDisks:
    (Empty(M.CurrentLinks))
&
Empty(CurrentNkcps)
&
for all (L:LineEntry) in Lines:
    (L.State = Free
     &
     L.CyclePosition = Available
     &
     L.AttachedVM = AuthProcess)
&
Empty(PendingRequests)
```

Invariant Assertions

#Nkcps <= #MaxNkcps
&
#VMs <= #MaxVMs
&
#Users <= #MaxUsers
&
InvariantsOfURPOwnedDevices
&
InvariantsOfNonsharableDrives
&
InvariantsOfSharableDrives
&
InvariantsOfSharedVolumes
&
. variantsOfMiniDisks
&
InvariantsOfCurrentNkcps
&
InvariantsOfLines
&
InvariantsOfPendingRequests
&
InvariantsOfUserDirectory

129

InvariantsOfURPOwnedDevices =

for all (U1,U2:URPOwnedDeviceEntry) in URPOwnedDevices:
   (U1.Raddr = U2.Raddr => U1 = U2)
&
for all (U:URPOwnedDeviceEntry) in URPOwnedDevices:
   (Dominates(U.MaxSecLevel,U.MinSecLevel)
     &
   DeviceType(Raddr) inset (Reader,Printer,Punch,TapeDrive))

```
InvariantsOfNonsharableDrives =

for all (NS1,NS2:NonsharableDriveEntry) in NonsharableDrives:
    (NS1.Raddr = NS2.Raddr => NS1 = NS2)
&
for all (NS:NonsharableDriveEntry) in NonsharableDrives:
    (Dominates(NS.MaxSecLevel,NS.MinSecLevel)
      &
    NS.State = Attached =>
        for some (N:NkcpEntry) in CurrentNkcps:
            (N.Process = NS.AttachedProcess
              &
            Dominates(NS.MaxSecLevel,N.Process)
              &
            Dominates(N.Process,NS.MinSecLevel)
              &
            for some (A:AttachedDeviceEntry) in N.AttachedDevices:
                (A.Raddr = NS.Raddr)))
```

131

```
InvariantsOfSharableDrives =

for all (S1,S2:SharableDriveEntry) in SharableDrives:
    (S1.Raddr = S2.Raddr => S1 = S2)
&
for all (S:SharableDriveEntry) in SharableDrives:
    (S.State = AttachedToSystem =>
        for some (V:SharedVolumeEntry) in SharedVolumes:
            (V.Volume = S.MountedVolume
            &
            V.State = Mounted
            &
            V.MountedDevice = S.Raddr
            &
            Dominates(S.SecLevel,V.SecLevel)))
```

132

InvariantsOfSharedVolumes =

for all (V1,V2:SharedVolumeEntry) in SharedVolumes:
    (V1.Volume = V2.Volume => V1 = V2)
&
for all (V:SharedVolumeEntry) in SharedVolumes:
    (V.State = Mounted =>
        for some (S:SharableDriveEntry) in SharableDrives:
            (S.Raddr = V.MountedDevice
            &
            S.MountedVolume = V.Volume
            &
            S.State = AttachedToSystem
            &
            Dominates(S.SecLevel,V.SecLevel)))

InvariantsOfMiniDisks =

```
for all (M1,M2:MiniDiskEntry) in MiniDisks:
    (M1.MDName = M2.MDName => M1 = M2)
&
for all (M:MiniDiskEntry) in MiniDisks:
    (for some (V:SharedVolumeEntry) in SharedVolumes:
        (V.Volume = M.ContainingVolume
        &
        Dominates(V.SecLevel,M.SecLevel))
    &
    M.Cylinders.2 > M.Cylinders.1
    &
    M.Cylinders.1 < #Cylinders(M.ContainingVolume)
    &
    M.Cylinders.2 <= #Cylinders(M.ContainingVolume)
    &
    for all (C:ProcessLinkEntry) in M.CurrentLinks:
        ((~Empty(C.Access))
        &
        for some (N:NkcpEntry) in CurrentNkcps:
            (N.Process = C.Process
            &
            for some (L:MDLinkEntry) in N.Links:
                (L.MDName = M.MDName
                &
                L.Access = C.Access)
            &
            for some (A:ACLEntry) in M.AccessControlList:
                (for some (V:VMEntry) in N.VMs:
                    (V.VMName = A.User))
            &
            Write inset C.Access =>
                N.Process = M.SecLevel))
    &
    ((~Empty(M.CurrentLinks)) =>
        for some (V:SharedVolumeEntry) in SharedVolumes:
            (V.Volume = M.ContainingVolume
            &
            V.State = Mounted
            &
            for some (S:SharableDriveEntry) in SharableDrives:
                (S.Raddr = V.MountedDevice
                &
                S.MountedVolume = V.Volume
                &
                S.State = AttachedToSystem)))
&
```

134

```
for all (A1,A2:ACLEntry) in M.AccessControlList:
    (A1.User = A2.User => A1 = A2)
&
for all (A:ACLEntry) in M.AccessControlList:
    (for some (D:DirectoryEntry) in UserDirectory:
        (D.UserId = A.User)
      &
      ~Empty(A.Access)))
```

135

InvariantsOfCurrentNkcps =

for all (N1,N2:NkcpEntry) in CurrentNkcps:
    (N1.Process = N2.Process => N1 = N2)
&
for all (N:NkcpEntry) in CurrentNkcps:
    (for all (VM1,VM2:VMEntry) in N.VMs:
        (VM1.VMName = VM2.VMName => VM1 = VM2)
    &
    for all (AD1,AD2:AttachedDeviceEntry) in N.AttachedDevices:
        (AD1.Raddr = AD2.Raddr => AD1 = AD2)
    &
    for all (L1,L2:MDLinkEntry) in N.Links:
        (L1.MDName = L2.MDName => L1 = L2)
    &
    for all (VM:VMEntry) in N.VMs:
        (for some (D:DirectoryEntry) in UserDirectory:
            (D.UserId = V.VMName
            &
            Dominates[D.MaxSecLevel,N.Process]
            &
            Dominates[N.Process,D.MinSecLevel])
        &
        (VM.Disconnected =>
                for all (L:LineEntry) in Lines:
                    ((L.AttachedVM = VM.VMName
                    &
                    L.RequestedSecLevel = N.Process
                    &
                    L.State = Attached) =>
                        L.Connection ~= Logon))
        &
        ((~VM.Disconnected) =>
                for some (L:LineEntry) in Lines:
                    (L.Laddr = VM.Laddr
                    &
                    L.AttachedVM = VM.VMName
                    &
                    L.RequestedSecLevel = N.Process
                    &
                    L.Connection = Logon
                    &
                    L.State inset (AttachValidation,Attached)))
        &
        for all (U:LineAddress) in VM.Users:
            (for some (L:LineEntry) in Lines:
                (L.Laddr = U
                &
                L.AttachedVM = VM.VMName
                &
                L.RequestedSecLevel = N.Process
                &
                L.Connection = Dial

136

```
                    &
                    L.State inset (AttachValidation,Attached))
                &
                U ~= VM.Laddr))
        &
        for all (AD:AttachedDeviceEntry) in N.AttachedDevices:
            (for some (NS:NonsharableDriveEntry) in NonsharableDrives:
                (NS.Raddr = AD.Raddr
                &
                Dominates(NS.MaxSecLevel,N.Process)
                &
                Dominates(N.Process,NS.MinSecLevel))
            &
            ~Empty(AD.Access))
        &
        for all (L:MDLinkEntry) in N.Links:
            (for some (M:MiniDiskEntry) in MiniDisks:
                (M.MDName = L.MDName
                &
                Dominates(N.Process,M.SecLevel)
                &
                for some (C:ProcessListEntry) in M.CurrentLinks:
                    (C.Process = N.Process
                    &
                    C.Access = L.Access)
                &
                for some (A:ACLEntry) in M.AccessControlList:
                    (for some (VM:VMEntry) in N.VMs:
                        (VM.VMName = A.User))
                &
                Write inset L.Access =>
                    M.SecLevel = N.Process)
            &
            ~Empty(L.Access)))
    &
    for all (N1,N2:NkcpEntry) in CurrentNkcps:
        (for all (AD1:AttachedDeviceEntry) in N1.AttachedDevices:
            (for all (AD2:AttachedDeviceEntry) in N2.AttachedDevices:
                (AD1.Raddr ~= AD2.Raddr)))
```

137

```
InvariantsOfLines =

for all (L1,L2:LineEntry) in Lines.
    (L1.Laddr = L2.Laddr => L1 = L2)
&
for all (':LineEntry) in Lines:
    (Dominates(L.MaxSecLevel,L.MinSecLevel)
     &
     L.State = AttachValidation =>
         L.CyclePosition inset (Retry,ReadInitialPassword,
                                ReadAccessPassword,HookingPeripherals,
                                NotifyingNkcp)
     &
     L.State = Attached =>
         L.CyclePosition inset (Attached,ReadLinkPassword)
     &
     L.State = Free =>
         L.CyclePosition inset (Disabled,Available,ReEnablePending)
     &
     L.State inset (AttachValidation,Attached) =>
         (Dominates(L.MaxSecLevel,L.RequestedSecLevel)
          &
          Dominates(L.RequestedSecLevel,L.MinSecLevel))
     &
     L.State = Attached =>
         (for some (N:NkcpEntry) in CurrentNkcps:
             (N.Process = L.RequestedSecLevel
              &
              for some (VM:VMEntry) in N.VMs:
                  (VM.VMName = L.AttachedVM
                   &
                   (L.Connection = Logon =>
                          (VM.Laddr = L.Laddr
                           &
                           ~VM.Disconnected))
                   &
                   (L.Connection = Dial =>
                          for some (U:LineAddress) in VM.Users:
                          (U = L.Laddr)))))
&
```

138

```
for all (L1,L2:LineEntry) in Lines:
    ((L1.Raddr ~= L2.Raddr
        &
      L1.State = Attached
        &
      L2.State = Attached
        &
      L1.AttachedVM = L2.AttachedVM
        &
      L1.RequestedSecLevel = L2.RequestedSecLevel
        &
      L1.Connection = Logon) =>
         L2.Connection = Dial)
```

139

InvariantsOfPendingRequests =

```
for all (P1,P2:PendingRequest) in PendingRequests:
   (P1.Msgid = P2.Msgid => P1 = P2)
&
for all (P:PendingRequest) in PendingRequests:
   (for all (R1,R2:ResponseSlot) in P.Responses:
      (R1.Respondent = R2.Respondent => R1 = R2)
   &
   for some (R:ResponseSlot) in P.Responses:
      (R.State = NoResponse)
   &
   ~Empty(P.Responses))
```

148

```
InvariantsOfUserDirectory =

for all (D1,D2:DirectoryEntry) in UserDirectory:
    (D1.Userid = D2.Userid => D1 = D2)
&
for all (D:DirectoryEntry) in UserDirectory:
    (Dominates(D.MaxSecLevel,D.MinSecLevel)
     &
     for all (DD1,DD2:DedicatedDeviceEntry) in D.DedicatedDevices:
        (DD1.Raddr = DD2.Raddr => DD1 = DD2)
     &
     for all (DD:DedicatedDeviceEntry) in D.DedicatedDevices:
        (DeviceType(DD.Raddr) inset (Reader,Printer,Punch,TapeDrive) =>
                (for some (U:URPOwnedDeviceEntry) in URPOwnedDevices:
                    (DD.Raddr = U.Raddr
                     &
                     DeviceType(DD.Raddr) = Reader =>
                        (DD.VolSecLevel = nil
                         &
                         DD.Access = (Read))
                     &
                     DeviceType(DD.Raddr) inset (Printer,Punch)=>
                        (DD.VolSecLevel = nil
                         &
                         DD.Access = (Write))
                     &
                     DeviceType(DD.Raddr) = TapeDrive =>
                        (Dominates(U.MaxSecLevel,DD.VolSecLevel)
                         &
                         Dominates(DD.VolSecLevel,U.MinSecLevel)
                         &
                         Dominates(D.MaxSecLevel,DD.VolSecLevel)
                         &
                         ~Empty(DD.Access))))
        &
        DeviceType(DD.Raddr) ~inset (Reader,Printer,Punch,TapeDrive) =>
                (for some (NS:NonsharableDriveEntry)
                             in NonsharableDrives:
                    (NS.Raddr = DD.Raddr
                     &
                     Dominates(NS.MaxSecLevel,DD.VolSecLevel)
                     &
                     Dominates(DD.VolSecLevel,NS.MinSecLevel)
                     &
                     Dominates(D.MaxSecLevel,DD.VolSecLevel)
                     &
                     ~Empty(DD.Access))))
    &
```

141

```
for all (L1,L2:MDLinkEntry) in D.Links:
    (L1.MDName = L2.MDName => L1 = L2)
&
for all (L:MDLinkEntry) in D.Links
    (for some (M:MiniDiskEntry) in MiniDisks:
        (M.MDName = L.MDName
         &
         for some (A:ACLEntry) in M.AccessControlList:
            (A.User = D.UserId
             &
             for all (AM:AccessModes) in L.Access:
                (AM inset A.Access))
         &
         Dominates(D.MaxSecLevel,M.SecLevel))
     &
     ~Empty(L.Access))
&
for all (AP1,AP2:AccessPasswordEntry) in D.AccessPasswords:
    (AP1.SecLevel = AP2.SecLevel => AP1 = AP2)
&
for all (AP:AccessPasswordEntry) in D.AccessPasswords:
    (Dominates(D.MaxSecLevel,AP.SecLevel)
     &
     Dominates(AP.SecLevel,D.MinSecLevel)))
```

{   Global Macros / Functions

```
macro EndAccessSequence(Line:LineEntry,Reason:ReasonTypes) =

        case Reason: ReasonTypes of
                IncorrectLogon: M <- "Forget it, Bub"

                ResourceFailure: M <-
                        "Requested security level not available
                        or not able to run another VM"

                SecurityViolation:

                MaxThresholdExceeded:

                NoNkcp:

                NoVM:

                TerminalClearanceMismatch:
        end

        KernelCall(SendMessage(
                WriteLine(L.Laddr,M),
                NetworkProcess))
        KernelCall(SendMessage(
                ClearLine(Line.Laddr),
                NetworkProcess))
        Line.State <- Free
        Line.CyclePosition <- ReEnablePending
        Line.AttachedVM <- AuthProcess
```

143

(

```
macro AddNkcpToSet [Process:ProcessName] =
        #Nkcps <- #Nkcps + 1
        CurrentNkcps <- Append [CurrentNkcps,
                                    <Process = Process,
                                     VMs = nil,
                                     AttachedDevices = nil,
                                     Links = nil>]
        KernelCall [SendMessage (AddNkcp [Process,Devices],URProcess)]
        KernelCall [SendMessage (AddNkcp [Process],OpProcess)]


macro Retry [L:LineEntry] =
        L.#Retries <- L.#Retries + 1
        if L.#Retries = #MaxRetries
           then EndAccessSequence [L,MaxThresholdExceeded]
           else KernelCall [SendMessage (
                       Retry [L.Laddr],
                       NetworkProcess)]
                L.CyclePosition <- Retry
        end


macro TryNotifyingNkcp [L:LineEntry] =

        if L.#AwaitingHooks = 0
           then L.Cycle Position <- NotifyingNkcp
                if ~L.LineDropped
                   then KernelCall [SendMessage (
                                ReDirectLine [
                                        L.Laddr,
                                        L.AttachedProcess],
                                NetworkProcess)]
                end
                KernelCall [SendMessage (NewVM [L],L.AttachedProcess)]
        end
```

144

```
MsgOp: process

        /* subdriver of AuthProcess,
                handling messages from OpProcess */

        given:  MsgId: MessageId
                Text: string

        entry:  just received message, Source = OpProcess

        action: if for some (P:PendingRequest) in PendingRequests:
                        (P.MsgId = MsgId)
                     then /* response to request */
                        error /* no requests to OpProcess */
                     else /* request from OpProcess */
                        case MsgName[Text] of
                           AUTOLOG:
                              OP1

                           ATTACH-RADDR:
                              DetermineRaddr[Text]
                              error on for all
                                        (NS:NonsharableDriveEntry)
                                            in NonsharableDrives:
                                                NS.Raddr ~= Raddr
                              OP3

                           DETACH-RADDR:
                              DetermineRaddr[Text]
                              if for some (NS:NonsharableDriveEntry)
                                            in NonsharableDrives:
                                        NS.Raddr = Raddr
                                 then OP4a
                                 else if for some
                                            (S:SharableDriveEntry)
                                                in SharableDrives:
                                                    S.Raddr = Raddr
                                         then OP4b
                                         else error
                                      end
                        end
```

145

```
                            VARY:
                                DetermineRaddr(Text)
                                if for some (S:SharableDriveEntry)
                                                in SharableDrives:
                                    S.Raddr = Raddr
                                then OP5a
                                else if for some
                                            (NS:NonsharableDriveEntry)
                                                in NonsharableDrives:
                                                NS.Raddr = Raddr
                                        then OP5b
                                        else error
                                    end
                            end

                            QUERY+DASD,
                            QUERY+LINES,
                            QUERY+GRAF,
                            QUERY+NAMES,
                            QUERY+USERS+X,
                            QUERY+ALL:
                                OP6a

                            QUERY+SYSTEM+RADDR,
                            QUERY+RADDR:
                                OP6b

                            QUERY+USERS+USERID,
                            QUERY+USERID:
                                OP6c

                            LOCATE+RADDR:
                                OP7

                            SHUTDOWN:
                                OP8

                            MapUserId:
                                OP2

                            other:
                                error
                        end
                end
        end MsgOp
```

MsgUR: process

```
        /* subdriver of AuthProcess,
                handling messages from URProcess */

        given:  Msgid: MessageId
                Text: string

        entry:  just received message, Source = URProcess. */

        action: if for some (P:PendingRequest) in PendingRequests:
                        P.MsgId = MsgId
                    then /* response to request */
                        error on P.Kind ~= Attach
                        case MsgName[Text] of
                            Attached:
                                UR3a

                            AttachFailed,
                            DeviceNotAvailable:
                                UR3b

                            other:
                                error
                        end
                    else /* request from URProcess */
                        case MsgName[Text] of
                            NeedNkcp:
                                UR1

                            MapUserId:
                                UR2

                            other:
                                error
                        end
                end

end MsgUR
```

147

MsgNot: process

/* subdriver of AuthProcess,
         handling messages from NetworkProcess */

given: MsgId: MessageId
         Text: string

entry: just received message, Source = NetworkProcess

action: if for some (P:PendingRequest) in PendingRequests:
                P.MsgId = MsgId
             then /* response to request */
                  case P.Kind:RequestCategory of
                      ClearLine,
                      ReDirectLine:
                          error on MsgName[Text] ~= LineStatus
                      NTWK1

                      WriteAndReadLine:
                        case MsgName[Text] of
                            LineInfo:
                                DetermineLaddr[Text]
                                let (L:LineEntry) in Lines:
                                    (L.Laddr = Laddr) in
                                case L.State:ActivityStatus of
                                    Attached:
                                        error on
                                            L.CyclePosition ~=
                                            ReadLinkPassword
                                    NTWK3

                                    AttachValidation:
                                        case L.CyclePosition:
                                            LineStatus of
                                        Retry:
                                            LGOL2

                                        ReadInitialPassword:
                                            LGOL3
                            if TEMP^L.CyclePosition =
                            PerformResourceChecks
                            then LGOL5
                                    if TEMP^L.CyclePosition =
                                    HookingPeripherals
                                    then LGOL6
                                        LGOL7
                                end
                                TryNotifyingNkcp[L]
                      end

148

```
                                    ReadAccessPassword:
                                        LGDL4
                    if TEMP"L.CyclePosition -
                        PerformResourceChecks
                        then LGDL5
                                if TEMP"L.CyclePosition -
                                    HookingPeripherals
                                    then LGDL6
                                        LGDL7
                            end
                            TryNotifyingNkcp(L)
                end

                                        other:
                                                error
                                        end

                                other:
                                        error
                            end

                        LineStatus:
                            NTWK1

                    other:
                            error
                end

            other:
                    error
          end
    else /* request from NetworkProcess */
        case MsgName(Text) of
                LineStatus:
                    NTWK1

                LineInfo:
                    LGDL1

                other:
                        error
                end
            end
end MsgNet
```

149

MsgNkcp: process

        /* subdriver of AuthProcess,
            handling messages from Nkcps */

    given: MsgId: MessageId
            Text: string
            Process: ProcessName

    entry: just received message, Source = Process */

    action: if for some (P:PendingRequest) in PendingRequests:
                    P.MsgId = MsgId
                then /* response to request */
                    case P.Kind:RequestCategory of
                        OpRequest:
                            error on MsgName[Text] ~=
                                    ResponseToOpRequest
                        ProcessResponse

                    NewVM,
                    ConnectVM,
                    NewUser,
                    NewOrConnectedVM:
                        DetermineLaddr[Text]
                        error on for all (L:LineEntry) in Lines:
                                    L.Laddr ~= Laddr
                        LGOL8
                        if TEMP"L.CyclePosition =
                            PerformResourceChecks
                            then LGOL5
                                    if TEMP"L.CyclePosition =
                                        HookingPeripherals
                                        then LGOL6
                                            LGOL7
                              · end
                                    TryNotifyingNkcp[L]
                    end

                RelinquishDevice:
                    error on MsgName[Text] ~= DetachDevice
                    NKCP6

                other:
                    error
            end

```
                    else /* request from Nkcp */
                        case MsgName(Text) of
                            Disconnect:
                                NKCP1

                            Logoff:
                                NKCP2

                            DropUser:
                                NKCP3

                            Link:
                                NKCP4

                            DetachDevice:
                                NKCP5

                            PurgeNkcp:
                                NKCP7

                            AccountingRecord:
                                NKCP8

                            other:
                                error
                        end
                end
end MsgNkcp
```

```
AuthDriver: process

        case HowWeGotHere of
            ExternalInterrupt:
                case InterruptSubType of
                    Message:
                        case Source of
                            OpProcess:          MsgOp
                            URProcess:          MsgUR
                            NetworkProcess:     MsgNet
                            Nkcp:               MsgNkcp
                            other:              /* anybody else talk
                                                   with AuthProcess? */

                    end

                    other: /* any other external interrupts? */
                end

            other: /* any other important interrupt classes? */
        end

        KernelCall [ReceiveInterrupts]
        KernelCall [ReleaseCPU]
end AuthDriver
```

NTWK1: Network process message re line status (both request and
response)

        given: Laddr: LineAddress
                CurrentLineStatus: LineCondition

        error on ~(for some (L:LineEntry) in Lines:
                    L.Laddr = Laddr)

        action: case L.State:ActivityStatus of
                    Free:
                        L.CyclePosition <- CurrentLineStatus
                        L.AttachedVM <- AuthProcess

                    Attached:
                        let (N:NkcpEntry) in CurrentNkcps:
                        (for some (V:VMEntry) in N.VMs:
                            (V.VMName = L.AttachedVM
                             &
                             ((L.Connection = Logon
                                  &
                                V.Laddr = L.Laddr)
                              or
                               (L.Connection = Dial
                                  &
                                for some (U:LineAddress) in V.Users:
                                    U = L.Laddr)))) in
                        let (V:VMEntry) in N.VMs:
                            (V.VMName = L.AttachedVM) in
                        if L.Connection = Logon
                          &
                          V.Laddr = L.Laddr
                          then V.Disconnected <- true
                                KernelCall[SendMessage(
                                        LineDisconnected[
                                                L.Laddr,
                                                V.VMName],
                                            N.Process)]
                                KernelCall[SendMessage(
                                        LineDisconnected[
                                                L.Laddr,
                                                V.VMName],
                                            OpProcess)]

                                    153

```
                        else let (U:LineAddress) in V.Users:
                             (U = L.Laddr) in
                             KernelCall[SendMessage(
                                     Dropped[V,U],
                                     N.Process)]
                             KernelCall[SendMessage(
                                     Dropped[V,U],
                                     OpProcess)]
                             V.Users <- Remove[V.Users,U]
             end
             L.State <- Free
             L.CyclePosition <- CurrentLineStatus
             L.AttachedVM <- AuthProcess

      AttachValidation:
         case L.CyclePosition:LineStatus of
              HookingPeripherals,
              NotifyingNkcp:
                  L.LineDropped <- true

              other:
                  L.State <- Free
                  L.CyclePosition <- CurrentLineStatus
                  L.AttachedVM <- AuthProcess
         end
   end
```

```
exit:   N"L.State =
                if L.State = Attached
                   |
                   (L.State = AttachValidation
                        &
                        L.CyclePosition ~inset (
                                HookingPeripherals,
                                NotifyingNkcp))
                then Free
                else L.State
                end
        N"L.CyclePosition =
                if L.State = Attached
                   |
                   (L.State = AttachValidation
                        &
                        L.CyclePosition ~inset (
                                HookingPeripherals,
                                NotifyingNkcp))
                then CurrentLineStatus
                else L.CyclePosition
                end
        N"L.AttachedVM =
                if L.State = Attached
                   |
                   (L.State = AttachValidation
                        &
                        L.CyclePosition ~inset (
                                HookingPeripherals,
                                NotifyingNkcp))
                then AuthProcess
                else L.AttachedVM
                end
        N"L.LineDropped =
                if L.State = AttachValidation
                        &
                        L.CyclePosition inset (
                            HookingPeripherals,
                            NotifyingNkcp)
                then true
                else L.LineDropped
                end
```

155

```
if L.State = Attached
   &
   L.Connection = Logon
   &
   L.Laddr = V.Laddr
   then N"V.Disconnected = true
   else N"V.Disconnected = V.Disconnected
end
if L.State = Attached
   &
   ~(L.Connection = Logon
        &
        V.Laddr = Laddr)
   &
   for some (U:LineAddress) in V.Users:
        (U = L.Laddr)
   then N"V.Users = Remove(V.Users,U)
   else N"V.Users = V.Users
end

if L.State = Attached
   then KernelCalled(SendMessage(OpProcess))
   else KernelCalled(SendMessage(N.Process))
end
```

156

LGOL1:   Network Process message, LOGON or DIAL request received

given: Laddr: LineAddress
         AttemptedCommand: AccessCategory
         UserId: VirtualMachineName
         RequestedSecLevel: ProcessName

error on ~(for some (L:LineEntry) in Lines:
         (L.Laddr = Laddr
          &
          L.State = Free
          &
          L.CyclePosition = Available))

action: if Dominates[L.MaxSecLevel,RequestedSecLevel]
          &
         Dominates[RequestedSecLevel,L.MinSecLevel]
         then L.#Retries <- 0
              L.#AwaitingHooks <- 0
              L.State <- AttachValidation
              L.Connection <- AttemptedCommand
              L.CyclePosition <- ReadInitialPassword
              L.AttachedVM <- UserId
              L.RequestedSecLevel <- RequestedSecLevel
              L.LineDropped <- false
              L.Msg <- nil
              KernelCall[SendMessage(
                       ReadInitialPassword[L.Laddr],
                       NetworkProcess)]
         else EndAccessSequence[L,
                               TerminalClearanceMismatch]
      end

157

LGDL2:   Network Process message (Retry of LOGON or DIAL)

        given: Laddr: LineAddress
                 UserId: VirtualMachineName
                 RequestedSecLevel: ProcessName

        entry: for some (L:LineEntry) in Lines:
                     (L.Laddr = Laddr
                        &
                      L.State = AttachValidation
                        &
                      L.CyclePosition = Retry)

        action: if Dominates(L.MaxSecLevel,RequestedSecLevel)
                       &
                   Dominates(RequestedSecLevel,L.MinSecLevel)
                   then L.CyclePosition <- ReadInitialPassword
                        L.AttachedVM <- UserId
                        L.RequestedSecLevel <- RequestedSecLevel
                        KernelCall(SendMessage(
                                ReadInitialPassword(L.Laddr),
                                NetworkProcess))
                   else EndAccessSequence(L,
                                          TerminalClearanceMismatch)
                 end

        exit: N"L.CyclePosition =
                        if Dominates(L.MaxSecLevel,RequestedSecLevel)
                              &
                         Dominates(RequestedSecLevel,L.MinSecLevel)
                         then ReadInitialPassword
                         else ReEnablePending
                        end
                 N"L.State =
                        if Dominates(L.MaxSecLevel,RequestedSecLevel)
                              &
                         Dominates(RequestedSecLevel,L.MinSecLevel)
                         then L.State /* AttachValidation */
                         else Free
                        end
                 N"L.AttachedVM =
                        if Dominates(L.MaxSecLevel,RequestedSecLevel)
                              &
                         Dominates(RequestedSecLevel,L.MinSecLevel)
                         then UserId
                         else AuthProcess
                        end

                                   158

```
            N"L.RequestedSecLevel =
                    if Dominates[L.MaxSecLevel,RequestedSecLevel]
                       &
                       Dominates[RequestedSecLevel,L.MinSecLevel]
                       then RequestedSecLevel
                       else L.RequestedSecLevel
                    end
            N"PendingRequests =
                    if Dominates[L.MaxSecLevel,RequestedSecLevel]
                       &
                       Dominates[RequestedSecLevel,L.MinSecLevel]
                       then Append[PendingRequests,Entry1]
                       else Append[PendingRequests,Entry2]
                    end

        if Dominates[L.MaxSecLevel,RequestedSecLevel]
           &
           Dominates[RequestedSecLevel,L.MinSecLevel]
           then KernelCalled[SendMessage(NetworkProcess)]
           else KernelCalled[SendMessage(NetworkProcess)]
                KernelCalled[SendMessage(NetworkProcess)]
        end

where Entry1 = <MsgId = new[MessageId],
               Kind = WriteAndReadLine,
               Command = Undefined,
               Responses = {<Respondent = NetworkProcess,
                             Text = nil,
                             State = NoResponse>}>
      Entry2 = <MsgId = new[MessageId],
               Kind = ClearLine,
               Command = Undefined,
               Responses = {<Respondent = NetworkProcess,
                             Text = nil,
                             State = NoResponse>}>
```

159

LGDL3: Userid, password, and requested security level validations


```
given:   Laddr: LineAddress
         Password: string

entry:   for some (L:LineEntry) in Lines:
              (L.Laddr = Laddr
               &
               L.State = AttachValidation
               &
               L.CyclePosition = ReadInitialPassword)

action: if for some (D:DirectoryEntry) in UserDirectory:
              (D.UserId = L.AttachedVM
               &
               (L.Connection = Logon =>
                    D.LogonPassword = Password)
               &
               (L.Connection = Dial =>
                    D.DialPassword = Password))
           then /* perform security level checks */
              if Dominates[D.MaxSecLevel,L.RequestedSecLevel]
                 &
                 Dominates[L.RequestedSecLevel,D.MinSecLevel]
                 then if for some (A:AccessPasswordEntry)
                                       in D.AccessPasswords:
                           (A.SecLevel = L.RequestedSecLevel)
                           then L.CyclePosition <-
                                   ReadAccessPassword
                                KernelCall[SendMessage(
                                   ReadAccessPassword[L.Laddr],
                                   NetworkProcess)]
                           else L.CyclePosition <-
                                   PerformResourceChecks
                 end
              else /* security violation */
                 EndAccessSequence[L,SecurityViolation]
           end
        else Retry[L]
     end
```

```
exit:   N"L.CyclePosition =
        if for some (D:DirectoryEntry) in UserDirectory:
            (D.UserId = L.AttachedVM
            &
            (L.Connection = Logon =>
                    D.LogonPassword = Password)
            &
            (L.Connection = Dial =>
                    D.DialPassword = Password))
        then if Dominates(D.MaxSecLevel,L.RequestedSecLevel)
            &
            Dominates(L.RequestedSecLevel,D.MinSecLevel]
            then if for some (A:AccessPasswordEntry)
                        in D.AccessPasswords:
                    (A.SecLevel = L.RequestedSecLevel)
                    then ReadAccessPassword
                    else PerformResourceChecks
                end
            else ReEnablePending
            end
        else if N"L.#Retries = #MaxRetries
            then ReEnablePending
            else Retry
            end
        end
        N"L.State =
        if (for some (D:DirectoryEntry) in UserDirectory:
            (D.UserId = L.AttachedVM
            &
            (L.Connection = Logon =>
                    D.LogonPassword = Password)
            &
            (L.Connection = Dial =>
                    D.DialPassword = Password))
        &
        ~(Dominates(D.MaxSecLevel,L.RequestedSecLevel)
            &
            Dominates(L.RequestedSecLevel,D.MinSecLevel)))
        |
        N"L.#Retries = #MaxRetries
        then Free
        else L.State
        end
        N"L.AttachedVM =
        if (for some (D:DirectoryEntry) in UserDirectory:
            (D.UserId = L.AttachedVM
            &
            (L.Connection = Logon =>
                    D.LogonPassword = Password)
            &
            (L.Connection = Dial =>'
                    D.DialPassword = Password))
            &
```

161

```
            ~(Dominates(D.MaxSecLevel,L.RequestedSecLevel)
                &
               Dominates(L.RequestedSecLevel,D.MinSecLevel)))
            |
            N"L.#Retries = #MaxRetries
            then AuthProcess
            else L.AttachedVM
      end
      N"L.#Retries =
      if ~(for some (D:DirectoryEntry) in UserDirectory:
            (D.UserId = L.AttachedVM
                &
               (L.Connection = Logon =>
                        D.LogonPassword = Password)
                &
               (L.Connection = Dial =>
                        D.DialPassword = Password)))
            then L.#Retries + 1
            else L.#Retries
      end
      N"PendingRequests =
      if for some (D:DirectoryEntry) in UserDirectory:
            (D.UserId = L.AttachedVM
                &
               (L.Connection = Logon =>
                        D.LogonPassword = Password)
                &
               (L.Connection = Dial =>
                        D.DialPassword = Password))
         then if Dominates(D.MaxSecLevel,L.RequestedSecLevel)
                &
               Dominates(L.RequestedSecLevel,D.MinSecLevel)
               then if for some (A:AccessPasswordEntry)
                                in D.AccessPasswords:
                        (A.SecLevel = L.RequestedSecLevel)
                        then Append(PendingRequests,Entry1)
                        else PendingRequests
                     end
               else Append(PendingRequests,Entry2)
            end
         else if N"L.#Retries = #MaxRetries
               then Append(PendingRequests,Entry2)
               else Append(PendingRequests,Entry1)
            end
      end
```

162

```
        if for some (D:DirectoryEntry) in UserDirectory:
            (D.UserId = L.AttachedVM
                &
            (L.Connection = Logon =>
                    D.LogonPassword = Password)
                &
            (L.Connection = Dial =>
                    D.DialPassword = Password))
        ·then If Dominates[D.MaxSecLevel,L.RequestedSecLevel]
                &
            Dominates[L.RequestedSecLevel,D.MinSecLevel]
            then ir for some (A:AccessPasswordEntry)
                        in D.AccessPasswords:
                    (A.SecLevel = L.RequestedSecLevel)
                    then KernelCalled[SendMessage(
                                    NetworkProcess)]
                end
            else KernelCalled[SendMessage(
                        NetworkProcess)]
                KernelCalled[SendMessage(
                        NetworkProcess)]
            end
        else if N"L.#Retries = #MaxRetries
            then KernelCalled[SendMessage(
                        NetworkProcess)]
                KernelCalled[SendMessage(
                        NetworkProcess)]
            else KernelCalled[SendMessage(
                        NetworkProcess)]
            end
        end

where Entry1 = <MsgId = new[MessageId],
            Kind = WriteAndReadLine,
            Command = Undefined,
            Responses = {<Respondent = NetworkProcess,
                            Text = nil,
                            State = NoResponse>}>
      Entry2 = <MsgId = new[MessageId],
            Kind = ClearLine,
            Command = Undefined,
            Responses = {<Respondent = NetworkProcess,
                            Text = nil,
                            State = NoResponse>}>
```

LGDL4:   Perform access password checks

        given: Laddr: LineAddress
                 AccessPassword: string

        entry: for some (L:LineEntry) in Lines:
                     (L.Laddr = Laddr
                      &
                      L.State = AttachValidation
                      &
                      L.CyclePosition = ReadAccessPassword)

        error on ~(for some (D:DirectoryEntry) in UserDirectory:
                     (D.UserId = L.UserId
                      &
                      for some (A:AccessPasswordEntry)
                                    in D.AccessPasswords:
                          A.SecLevel = L.RequestedSecLevel)))

        action: if A.Password = AccessPassword
                   then L.CyclePosition <- PerformResourceChecks
                   else KernelCall(SendMessage(
                                        SecViol (L.Laddr,
                                                 L.AttachedVM,
                                                 RequestedSecLevel),
                                        OpProcess))
                        Retry(L)
                        end
              end

        exit: N"L.CyclePosition =
                        if A.Password = AccessPassword
                          then PerformResourceChecks
                          else if L.#Retries + 1 = #MaxRetries
                                  then ReEnablePending
                                  else Retry
                              end
                     end
                 N"L.#Retries =
                        if A.Password ~= AccessPassword
                          then L.#Retries + 1
                          else L.#Retries
                     end
                 N"L.State =
                        if A.Password ~= AccessPassword
                          &
                          N"L.#Retries = #MaxRetries
                          then Free
                          else L.State
                        end

                                    164

```
                N"L.AttachedVM =
                        if A.Password ~= AccessPassword
                                &
                                N"L.#Retries = #MaxRetries
                                then AuthProcess
                                else L.AttachedVM
                        end
                N"PendingRequests =
                        if A.AccessPassword = Password
                                then PendingRequests
                                else if N"L.#Retries = #MaxRetries
                                        then Append[PendingRequests,Entry1]
                                        else Append[PendingRequests,Entry2]
                                end
                end

                if A.Password ~= AccessPassword
                        then KernelCalled[SendMessage(OpProcess)]
                                if L.#Retries + 1 = #MaxRetries
                                        then KernelCalled[SendMessage(
                                                        NetworkProcess)]
                                                KernelCalled[SendMessage(
                                                        NetworkProcess)]
                                        else KernelCalled[SendMessage(
                                                        NetworkProcess)]
                                end
                end

        where Entry1 = <MsgId = new[MessageId],
                        Kind = ClearLine,
                        Command = Undefined,
                        Responses = (Respondent = NetworkProcess,
                                        Text = nil,
                                        State = NoResponse>]>
                Entry2 = <MsgId = new[MessageId],
                        Kind = WriteAndReadLine,
                        Command = Undefined.
                        Responses = (Respondent = NetworkProcess,
                                        Text = nil,
                                        State = NoResponse>]>
```

165

LGDL5:   Perform resource checks

        given: L:LineEntry in Lines
                 D:DirectoryEntry in UserDirectory

        error on ~(D.UserId = L.AttachedVM
                  &
                  ~ ShuttingDown)

        action: If L.Connection = Logon
                   then if for all (N:NkcpEntry) in CurrentNkcps:
                           N.Process ~= L.RequestedSecLevel
                        then if #Nkcps < #MaxNkcps
                               then CreateNkcp(L.RequestedSecLevel)
                               else EndAccessSequence(L,NoNkcp)
                            end
                   end
                   let (N:NkcpEntry) in CurrentNkcps:
                      (N.Process = L.RequestedSecLevel) in
                   if for all (V:VMEntry) in N.VMs:
                           V.VMName ~= L.AttachedVM
                        then KernelCall(CreateVM(L.AttachedVM,
                                        N.Process))
                           if OK
                              then #VMs <- #VMs + 1
                                   #Users <- #Users + 1
                                   N.VMs <- Append(N.VMs,
                                           <VMName =
                                               L.AttachedVM,
                                            Laddr = L.Laddr,
                                            Disconnected = false,
                                            Users = nil,
                                            Msg = nil>)
                                   L.CyclePosition <- HookingPeripherals
                              else EndAccessSequence(L,NoVM)
                                   KernelCall(SendMessage(
                                           PurgeIfAble,N.Process))
                        end

166

```
                   else let (V:VMEntry) in N.VMs:
                         (V.VMName = L.AttachedVM) in
                      if V.Disconnected
                         then V.Laddr <- L.Laddr
                              V.Disconnected <- false
                              #Users <- #Users + 1
                              KernelCall[SendMessage(
                                  ReDirectLine[L.Laddr,
                                              N.Process],
                                  NetworkProcess)]
                              KernelCall[SendMessage(
                                  ConnectedVM[L],
                                  N.Process)]
                              L.CyclePosition <-
                                      NotifyingNkcp
                         else Retry[L]
                     end
             end
         else if L.Connection = Dial
              then if for some (N:NkcpEntry)
                         in CurrentNkcps:
                         (N.Process = L.RequestedSecLevel
                         &
                         for some (V:VMEntry) in N.VMs:
                             (V.VMName = L.AttachedVM))
                      then KernelCall[SendMessage(
                              ReDirectLine[
                                      L.Laddr,
                                      N.Process],
                                  NetworkProcess)]
                              KernelCall[SendMessage(
                                  NewUser[L],
                                  N.Process)]
                              L.CyclePosition <-
                                      NotifyingNkcp
                              #Users <- #Users + 1
                              V.Users <- Append[V.Users,
                                              L.Laddr]
                         else Retry[L]
                     end
                 end
     end
```

LGDL6:   Attach Dedicated Devices


```
given:  Laddr: LineAddress

entry:  for some (L:LineEntry) in Lines:
             (L.Laddr = Laddr
               &
              L.State = AttachValidation
               &
              L.CyclePosition = HookingPeripherals)

error on ~(for some (N:NkcpEntry) in CurrentNkcps:
                 (N.Process = L.AttachedProcess)
               &
              for some (D:DirectoryEntry) in UserDirectory:
                 (D.UserId = L.AttachedVM))

action:
for all (A:DedicatedDeviceEntry) in D.DedicatedDevices:
  if for some (B:URPOwnedDeviceEntry) in URPOwnedDevices:
        (A.Raddr = B.Raddr)
     then /* URProcess controls allocation */
          if Dominates(B.MaxSecLevel,N.Process)
            &
           Dominates(N.Process,B.MinSecLevel)
           then KernelCall(SendMessage(
                         Attach(A.Raddr,N.Process),
                         URProcess))
                L.#AwaitingHooks <- L.#AwaitingHooks + 1
             else L.Msg <- Concat(L.Msg,Unavail(A.Raddr))
          end
```

```
else /* AuthProcess controls allocation */
    let (B:NonsharableDriveEntry) in NonsharableDrives:
        (A.Raddr = B.Raddr) in
    if B.State = Available
        &
        Dominates(B.MaxSecLevel,N.Process)
        &
        Dominates(N.Process,B.MinSecLevel)
        &
        Dominates(B.MaxSecLevel,A.VolSecLevel)
        &
        Dominates(A.VolSecLevel,B.MinSecLevel)
        &
        Dominates(N.Process,A.VolSecLevel)
        then B.Access <- A.Access
            if Write inset B.Access
                &
                A.VolSecLevel ~= N.Process
                then B.Access <- Remove(B.Access,Write)
            end

            KernelCall(GrantAccess(
                    B.Raddr,
                    N.Process,
                    B.Access))
            if OK
                then N.AttachedDevices <- Append(
                            N.AttachedDevices,
                            <Raddr = B.Raddr,
                             Access = B.Access>)
                    B.State <- AttachedToUser
                    B.AttachedProcess <- N.Process
                    L.Msg <- Concat(L.Msg,
                                    Avail(A.Raddr,Access))
                else L.Msg <- Concat(L.Msg,
                                     Unavail(A.Raddr))
            end
        else L.Msg <- Concat(L.Msg,Unavail(A.Raddr))
    end
end
```

LGDL7:   Perform Links at Logon

       given: Laddr: LineAddress

       entry: for some (L:LineEntry) in Lines:
              (L.Laddr = Laddr
                &
              L.State = AttachValidation
                &
              L.CyclePosition = HookingPeripherals)

       error on ~(for some (N:NkcpEntry) in CurrentNkcps:
                (N.Process = L.AttachedProcess)
            &
             for some (D:DirectoryEntry) in UserDirectory:
                (D.UserId = L.AttachedVM))

     action:

170

LGOL8:   Response to message to NKCP re new VM

        given: VM: VirtualMachineName
                 Process: ProcessName
                 Laddr: LineAddress

        entry: for some (L:LineEntry) in Lines:
                     (L.Laddr = Laddr)

        error on ~(L.AttachedVM = VM
                       &
                     L.State = AttachValidation
                       &
                     L.CyclePosition = NotifyingNkcp)

        action: if Responded(Process)
                     then L.State <- Attached
                         L.CyclePosition <- Attached
                         if L.LineDropped
                             then KernelCall[SendMessage(
                                         ClearLine(L.Laddr),
                                         NetworkProcess)]
                         end
                     else if for some (N:NkcpEntry) in CurrentNkcps:
                                 (N.Process = Process)
                             then if L.Connection = Logon
                                     then KernelCall[SendMessage(
                                             NewOrConnectedVM[L],
                                             N.Process)]
                                     else KernelCall[SendMessage(
                                             NewUser[L],
                                             N.Process)]
                             end
                         else if L.LineDropped
                                 then KernelCall[SendMessage(
                                             ClearLine(L.Laddr),
                                             NetworkProcess)]
                                     L.State <- Free
                                 else L.CyclePosition <-
                                             PerformResourceChecks
                             end
                     end
                 end

```
exit:    N"L.State =
                   if Responded(Process)
                     then Attached
                     else if for all (N:NkcpEntry)
                                        in CurrentNkcps:
                                   (N.Process ~= Process)
                                   &
                                   L.LineDropped
                              then Free
                              else L.State
                          end
             end
         N"L.CyclePosition =
                   if Responded(Process)
                     then Attached
                     else if for some (N:NkcpEntry)
                                        in CurrentNkcps:
                                   (N.Process = Process)
                                   |
                                   L.LineDropped
                              then L.CyclePosition
                              else PerformResourceChecks
                          end
             end

         if Responded(Process)
            then KernelCalled(SendMessage(NetworkProcess))
            else if for some (N:NkcpEntry) in CurrentNkcps:
                           (N.Process = Process)
                      then KernelCalled(SendMessage(N.Process))
                      else if L.LineDropped
                              then KernelCalled(SendMessage(
                                             NetworkProcess))
                          end
                 end
         end
```

172

NKCP1:   Disconnect

        given: Process: ProcessName
                 VM: VirtualMachineName
                 Laddr: LineAddress
                 LineAction: string

        error on ~(for some (N:NkcpEntry) in CurrentNkcps:
                    (N.Process = Process)
                 &
                 for some (L:LineEntry) in Lines:
                    (L.Laddr = Laddr))

        action: if for some (V:VMEntry) in N.VMs:
                    (V.VMName = VM
                     &
                     V.Laddr = Laddr
                     &
                     V.Disconnected = false)
                 then N.V.Disconnected <- true
                    if LineAction = 'hold'
                        then KernelCall[SendMessage(
                                        ReDirectLine[L.Laddr,
                                                     AuthProcess],
                                        NetworkProcess)]
                        else KernelCall[SendMessage(
                                        ClearLine[L.Laddr],
                                        NetworkProcess)]
                    end
                    L.State <- Free
                    L.CyclePosition <- ReEnablePending
                    #Users <- #Users - 1
                 else /* ignore */
                 end

        exits: N"L.State =
                    if for some (V:VMEntry) in N.VMs:
                        (V.VMName = VM
                         &
                         V.Laddr = Laddr
                         &
                         ~V.Disconnected)
                    then Free
                    else L.State
                    end

                                173

```
N"L.CyclePosition =
        if for some (V:VMEntry) in N.VMs:
                (V.VMName = VM
                     &
                     V.Laddr = Laddr
                     &
                     ~V.Disconnected)
              then ReEnablePending
              else L.CyclePosition
        end
N"#Users =
        if for some (V:VMEntry) in N.VMs:
                (V.VMName = VM
                     &
                     V.Laddr = Laddr
                     &
                     ~V.Disconnected)
              then #Users - 1
              else #Users
        end
N"PendingRequests =
        if for some (V:VMEntry) in N.VMs:
                (V.VMName = VM
                     &
                     V.Laddr = Laddr
                     &
                     ~V.Disconnected)
              then Append[PendingRequests,Entry]
              else PendingRequests
        end
if for some (V:VMEntry) in N.VMs:
        (V.VMName = VM
             &
             V.Laddr = Laddr
             &
             ~V.Disconnected)
     then N"V.Disconnected = true
end

KernelCalled[SendMessage(NetworkProcess)]

where Entry =   <MsgId = new[MessageId],
                Kind =  if LineAction = 'hold'
                             then ReDirectLine
                             else ClearLine
                        end,
              Command = Undefined,
              Responses = [Respondent = NetworkProcess,
                              Text = nil,
                              State = NoResponse>]>
```

NKCP2: Logoff


```
given:  Process: ProcessName
        VM: VirtualMachineName
        LineAction: string
        ReasonForLogoff: LogoffReasons

error on ~(for some (N:NkcpEntry) in CurrentNkcps:
            (N.Process = Process
             &
             for some (V:VMEntry) in N.VMs:
                (V.VMName = VM)))

action: KernelCall[DestroyVM(V.VMName)]
        if OK
           then #Users <- #Users - 1
                #VMs <- #VMs - 1
                if V.Disconnected
                   then case ReasonForLogoff:LogoffReasons of
                              UserChoice:
                                  error

                              Forced:
                                  KernelCall[SendMessage(
                                      ForcedLogoff[V.VMName],
                                      OpProcess)]

                              Disconnected:
                                  KernelCall[SendMessage(
                                      DisconnLogoff[V.VMName],
                                      OpProcess)]
                       end
                   else if LineAction = 'hold'
                           then KernelCall[SendMessage(
                                   ReDirectLine[
                                       V.Laddr,
                                       AuthProcess],
                                   NetworkProcess)]
                           else KernelCall[SendMessage(
                                   ClearLine[V.Laddr],
                                   NetworkProcess)]
                end
                L.State <- Free
                L.CyclePosition <- ReEnablePending
```


175

```
                              case ReasonForLogoff:LogoffReasons of
                              UserChoice:
                                      KernelCall [SendMessage(
                                              Logoff [
                                                      V.VMName,
                                                      V.Laddr,
                                                      #Users],
                                              OpProcess)]

                              Forced:
                                  KernelCall [SendMessage(
                                              ForcedLogoff [V.VMName],
                                              OpProcess)]

                              Disconnected:
                                      error

                          end
                      end
                      for all (U:LineAddress) in V.Users:
                          KernelCall [SendMessage(
                                  ClearLine [U],
                                  NetworkProcess)]
                          KernelCall [SendMessage(
                                  Dropped [V,U],
                                  OpProcess)]
                          #Users <- #Users - 1
                          let (Line:LineEntry) in Lines:
                              (Line.Laddr = U) in
                          Line.State <- Free
                          Line.CyclePosition <- ReEnablePending
                      VMs <- Remove [VMs,V]
                      KernelCall [SendMessage(
                                  SystemResourceUse [N.Process,VM],
                                  AcntProcess)]
                  else KernelCall [SendMessage(
                              VMNotDestroyed [VM],
                              N.Process)]

              end

  exit:    N"#Users =
                  if DestroyedVM
                      then #Users - C"V.Users - 1
                      else #Users
                  end
           N"#VMs =
                  if DestroyedVM
                      then #VMs - 1
                      else #VMs
                  end
```

176

```
N"N.VMs =
        if DestroyedVM
            &
            ~ V.Disconnected
            then Remove(N.VMs,V)
            else N.VMs
        end
N"PendingRequests =
        if DestroyedVM
            &
            ~ V.Disconnected
            then if LineAction = 'hold'
                    then union of(
                            Append(PendingRequests,Entry1),
                            set: for all (U:LineAddress)
                                            in V.Users:
                                    Entry2)
                    else union of (
                            Append(PendingRequests,Entry2),
                            set: for all (U:LineAddress)
                                            in V.Users:
                                    Entry2)
                end
            else PendingRequests
        end
for all (U:LineAddress) in V.Users:
    for some (Line:LineEntry) in Lines:
        (Line.Laddr = U
        &
        N"Line.State = Free
        &
        N"Line.CyclePosition = ReEnablePending)

KernelCalled(DestroyVM(V.VMName))
if DestroyedVM
    then if V.Disconnected
            then KernelCalled(SendMessage(OpProcess))
            else KernelCalled(SendMessage(OpProcess))
                KernelCalled(SendMessage(
                        NetworkProcess))
                KernelCalled(SendMessage(
                        NetworkProcess))
                KernelCalled(SendMessage(
                        AcntProcess))
                for all (U:LineAddress) in V.Users:
                    (KernelCalled(SendMessage(
                        NetworkProcess))
                    KernelCalled(SendMessage(
                        NetworkProcess))
    else KernelCalled(SendMessage(N.Process))
end
```

177

```
where Entry1 = <MsgId = new(MessageId),
                Kind = ReDirectLine,
                Command = Undefined,
                Responses = (Respondent = NetworkProcess,
                             Text = nil,
                             State = NoR; ponse>)>
      Entry2 = <MsgId = new(MessageId),
                Kind = ClearLine,
                Command = Undefined,
                Responses = (Respondent = NetworkProcess,
                             Text = nil,
                             State = NoResponse>)>
```

178

OP1:   Autolog

```
given:   UserId: VirtualMachineName
         RequestedSecLevel: ProcessName
         Password: string
         AccessPassword: string

error on ~(for some (D:DirectoryEntry) in UserDirectory:
             (D.UserId = UserId
              &
              D.LogonPassword = Password
              &
              Dominates(D.MaxSecLevel,RequestedSecLevel)
              &
              Dominates(RequestedSecLevel,D.MinSecLevel)
              &
              D.IplDefined = true
              &
              for all (A:AccessPasswordEntry)
                                in D.AccessPasswords:
                  ((A.SecLevel = RequestedSecLevel) =>
                          (A.Password = AccessPassword)))
           &
           ~ ShuttingDown
           &
           #VMs < #MaxVMs)

action: if for all (N:NkcpEntry) in CurrentNkcps:
           (N.Process ~= RequestedSecLevel)
           then if #Nkcps < #MaxNkcps
                   &
                   #VMs < #MaxVMs
                   then KernelCall (CreateProcess(
                                AddressSpaceSize,
                                CodeSize,Code))
                        if OK
                           then AddNkcpToSet(
                                   RequestedSecLevel)
                           else KernelCall (SendMessage(
                                   NoNkcp,OpProcess))
                                exit
                        end
                   else KernelCall (SendMessage(NoNkcp,
                                OpProcess))
                        exit
                end
        end
        let (N:NkcpEntry) in CurrentNkcps:
            (N.Process = RequestedSecLevel) in
        KernelCall (CreateVM(UserId,N.Process))
```

179

```
if OK
    then #VMs <- #VMs + 1
         #Users <- #Users + 1
         N.VMs <- Append[N.VMs, <VMName = Userid,
                                  Laddr = ??,
                                  Disconnected = true,
                                  Users = nil>]
         KernelCall[SendMessage(
                 Autolog[Userid,N.Process],
                 OpProcess)]
         /* AttachDevices[N,D]
            PerformLinks[N,D] */
         KernelCall[SendMessage(NewVM[],N.Process)]
    else KernelCall[SendMessage(NoVMSpace,OpProcess)]
         KernelCall[SendMessage(
                 PurgeIfAble,N.Process)]
end
```

UR2 and OP2: Map user id

        given: UserId: VirtualMachineName
                 Requester: ProcessName

        entry: Requester inset (OpProcess, URProcess)

        action: if for some (N:NkcpEntry) in CurrentNkcps:
                      for some (V:VMEntry) in N.VMs:
                          (V.VMName = UserId)
                     then KernelCall(SendMessage(UserIdMapped(
                                          UserId, N.Process),
                                          Requester))
                      else KernelCall(SendMessage(UserIdMapped(
                                          UserId, nil),
                                          Requester))
                 end

        exit: KernelCalled(SendMessage(Requester))

181

OP4b: Detach of shared device (by operator)

        given: Raddr: DeviceAddress

        entry: for some (S:SharableDriveEntry) in SharableDrives:
                    S.Raddr = Raddr

        action: case S.State:SharableDriveStatus of
                    Available:
                        KernelCall [SendMessage(
                            Detached[S.Raddr],
                            OpProcess)]

                    OffLine:
                        KernelCall [SendMessage(
                            OffLine[S.Raddr],
                            OpProcess)]

                    AttachedToSystem:
                        if for some (M:MiniDiskEntry) in MiniDisks:
                            (M.ContainingVolume = S.MountedVolume
                                &
                            ~Empty[M.ProcessLinks])
                        then KernelCall [SendMessage(
                                NotCurrentlyDetachable[S.Raddr],
                                OpProcess)]
                        else KernelCall [IsDeviceReleasable(S.Raddr)]
                            if OK
                                then KernelCall [ReleaseDevice(
                                        S.Raddr)]
                                    if OK
                                        then KernelCall [
                                                SendMessage(
                                                  Detached[S.Raddr],
                                                  OpProcess)]
                                            S.State <- Available
                                        let (V:SharedVolumeEntry)
                                            in SharedVolumes:
                                            (V.MountedDevice
                                                =
                                            S.MountedVolume) in
                                            V.State <- NotMounted
                                        else KernelCall [
                                                SendMessage(
                                                  NotDetached[
                                                    S.Raddr],
                                                  OpProcess)]

                            end

182

```
                                    else KernelCall[SendMessage(
                                            NotCurrentlyDetachable[
                                                    S.Raddr],
                                            OpProcess)]
                            end
                    end
            end

    exit:   N"S.State =
                    if S.State = AttachedToSystem
                        &
                        for all (M:MiniDiskEntry) in MiniDisks:
                            (M.ContainingVolume = S.MountedVolume ->
                                        Empty[M.CurrentLinks])
                        &
                        DeviceIsReleasable
                        &
                        DeviceReleased
                        then Available
                        else S.State
                    end
            if S.State = AttachedToSystem
                &
                for all (M:MiniDiskEntry) in MiniDisks:
                    (M.ContainingVolume = S.MountedVolume ->
                        Empty[M.CurrentLinks])
                &
                DeviceIsReleasable
                &
                DeviceReleased
                    then for some (V:SharedVolumeEntry) in SharedVolumes:
                            (V.MountedDevice = S.MountedVolume
                            &
                            N"V.State = NotMounted
                            &
                            N"V.MountedDevice = nil)

            KernelCalled[SendMessage(OpProcess)]
            if S.State = AttachedToSystem
                then if for all (M:MiniDiskEntry) in MiniDisks:
                            (M.ContainingVolume = S.MountedVolume ->
                                    Empty[M.CurrentLinks])
                        then KernelCalled[IsDeviceReleasable]
                            if DeviceIsReleasable
                                then KernelCalled[RelsaseDevice]
                            end
                    end
            end
```

183

OP4a: Detach of nonshared device (by operator)

        given: Raddr: DeviceAddress

        entry: for some (NS:NonsharableDriveEntry)
                            in NonsharableDrives:
                      NS.Raddr = Raddr

        action: case NS.State: DriveStatus of
                    OffLine:
                        KernelCall [SendMessage (
                                OffLine [NS.Raddr],
                                OpProcess)]

                    DetachPending:
                        /* ignore */

                    Available:
                        KernelCall [SendMessage (
                                Detached [NS.Raddr],
                                OpProcess)]

                    AttachedToUser:
                        KernelCall [SendMessage (
                                RelinquishDevice [NS.Raddr],
                                NS.AttachedProcess)]
                        NS.State <- DetachPending
                end

```
exit:   N"NS.State =
                if NS.State = AttachedToUser
                   then DetachPending
                   else NS.State
              end
        N"PendingRequests =
                if NS.State = AttachedToUser
                   then Append(PendingRequests,Entry)
                   else PendingRequests
              end

        if NS.State = AttachedToUser
           then KernelCall(SendMessage(NS.AttachedProcess)]
           else if NS.State inset (OffLine, Available)
                   then KernelCalled(SendMessage(OpProcess)]
                 end
        end

where Entry =   <MsgId = new(MessageId),
                Kind = RelinquishDevice,
                Command = DETACH+RADDR,
                Responses = (<Respondent = NS.AttachedProcess,
                                Text = nil,
                                State = NoResponse>)>
```

185

OPS5a: Vary (both online and offline) of shared device

        given: Raddr: DeviceAddress
                 Parameter: string

        entry: for some (S:SharableDriveEntry) in SharableDrives:
                      S.Raddr = Raddr

        error on Parameter ~inset ('online, 'offline')

        action: if Parameter = 'online'
                    then error on S.State ~= OffLine
                         S.State <- Available
                         KernelCall(SendMessage(
                                 OnLine(S.Raddr),
                                 OpProcess))
                    else /* parameter = 'offline' */
                         error on S.State ~= AttachedToSystem
                         S.State <- OffLine
                         KernelCall(SendMessage(
                                 OffLine(S.Raddr),
                                 OpProcess))

                end

        exit: N"S.State =
                         if Parameter = 'online'
                             then Available
                             else OffLine
                         end

        KernelCalled(SendMessage(OpProcess))

186

OP5b: Vary (both online and offline) of nonshared device

        given: Raddr: DeviceAddress
                 Parameter: string

        entry: for some (NS:NonsharableDriveEntry)
                              in NonsharableDrives:
                     NS.Raddr = Raddr

        error on Parameter ~inset ('online', 'offline')

        action: if Parameter = 'online'
                    then error on NS.State ~= OffLine
                         NS.State <- Available
                         KernelCall[SendMessage(
                                 OnLine(NS.Raddr),
                                 OpProcess)]
                    else /* parameter = 'offline' */
                         error on NS.State ~inset (OffLine,
                                                    Available)

                         NS.State <- OffLine
                         KernelCall[SendMessage(
                                 OffLine(NS.Raddr),
                                 OpProcess)]
                end

        exit: N"NS.State =
                         if Parameter = 'online'
                            then Available
                            else OffLine
                         end

                KernelCalled[SendMessage(OpProcess)]

                                187

OP6a: QUERY, with parameters:
         DASD
         LINES
         GRAF
         ALL
         NAMES
         USERS with no further parameter

    entry: true

    action: use table information to create message
        KernelCall[SendMessage(Info,OpProcess)]

    exit: KernelCalled[SendMessage(OpProcess)]

188

OP6b: QUERY, with parameters:
                    raddr
                    SYSTEM raddr

        given: Raddr: DeviceAddress

        error on ~(for some (S:SharableDriveEntry) in SharableDrives:
                    (S.Raddr = Raddr)
                |
                for some (NS:NonsharableDriveEntry)
                                    in NonsharableDrives:
                    (NS.Raddr = Raddr))

        action: use table information to create message
                KernelCall[SendMessage(Info,OpProcess)]

        exit: KernelCalled[SendMessage(OpProcess)]

189

```
OP6c: QUERY, with parameters:
              USERS userid
              userid

      given:  UserId: VirtualMachineName

      entry:  true

      action: if for some (N:NkcpEntry) in CurrentNkcps:
                    for some (V:VMEntry) in N.VMs:
                        V.VMName = UserId
                 then use table information to create message
                     KernelCall(SendMessage(Info,OpProcess))
                 else KernelCall(SendMessage(NoSuchUser(UserId),
                                             OpProcess))

              end

      exit:   KernelCalled(SendMessage(OpProcess))
```

190

OP7:   LOCATE←RADDR

given: Raddr: DeviceAddress

entry: for some (NS:NonsharableDriveEntry)
                        in NonsharableDrives:
              NS.Raddr = Raddr

action: if NS.State = AttachedToUser
            then KernelCall(SendMessage(
                        Locate(NS.Raddr),
                        N.Process))
            else KernelCall(SendMessage(
                        DeviceNotOwned(D.Raddr),
                        OpProcess))
         end

exit: N"PendingRequests =
                if NS.State = AttachedToUser
                    then Append(PendingRequests,
                                    <MsgId = new(MessageId),
                                    Kind = OpRequest,
                                    Command = LOCATE←RADDR,
                                    Responses = (
                            <Respondent = NS.AttachedProcess,
                             Text = nil,
                             State = NoResponse>)>)
                    else PendingRequests
                end

         if NS.State = AttachedToUser
            then KernelCalled(SendMessage(NS.AttachedProcess))
            else KernelCalled(SendMessage(OpProcess))
         end

191

OP8: Shutdown

     entry: true

     action: ShuttingDown <- true

     exit: N"ShuttingDown = true

OP3:   Attach (nonsharable disk drive) Device

given: Raddr: DeviceAddress
         Process: ProcessName
         VolSecLevel: ProcessName
         Access: set of AccessModes

entry: for some (NS:NonsharableDriveEntry)
                            in NonsharableDrives:
                    (NS.Raddr = Raddr)

error on ~(for some (N:NkcpEntry) in CurrentNkcps:
                (N.Process = Process
                 &
                 for all (A:AttachedDeviceEntry)
                                    in N.AttachedDevices:
                     (A.Raddr ~= Raddr)
                 &
                 NS.State = Free
                 &
                 Dominates[N.Process,VolSecLevel]
                 &
                 Dominates[NS.MaxSecLevel,N.Process]
                 &
                 Dominates[N.Process,NS.MinSecLevel]
                 &
                 Dominates[NS.MaxSecLevel,VolSecLevel]
                 &
                 Dominates[VolSecLevel,NS.MinSecLevel])
            &
            ~Empty[Access]
            &
            ~ ShuttingDown)

193

```
action: NS.Access <- Access
        if Write inset NS.Access
           &
           VolSecLevel ~= N.Process
           then NS.Access <- Remove(NS.Access,Write)
        end
        if Empty(NS.Access)
           then KernelCall(SendMessage(
                        NoAccess(NS.Raddr,N.Process),
                        OpProcess))
           else KernelCall(GrantAccess(
                        NS.Raddr,
                        N.Process,
                        NS.Access))
                if GrantedAccess
                   then KernelCall(SendMessage(
                                Attached(NS.Raddr),
                                N.Process))
                        KernelCall(SendMessage(
                                Attached(
                                        NS.Raddr,
                                        NS.AttachedProcess),
                                OpProcess))
                        NS.State <- Attached
                        NS.AttachedProcess <- N.Process
                        N.AttachedDevices <- Append(
                                N.AttachedDevices,
                                <Raddr = NS.Raddr,
                                 Access = NS.Access>)
                   else KernelCall(SendMessage(
                                CannotAttach(
                                        NS.Raddr,
                                        N.Process),
                                OpProcess))
                end
        end
```

194

```
exit:    N"NS.Access =
                    if Write inset Access
                        &
                        VolSecLevel ~= N.Process
                        then Remove[Access,Write]
                        else Access
                    end
          N"NS.State =
                    if ~Empty[N"NS.Access]
                        &
                        GrantedAccess
                        then Attached
                        else NS.State
                    end
          N"NS.AttachedProcess =
                    if ~Empty[N"NS.Access]
                        &
                        GrantedAccess
                        then N.Process
                        else NS.AttachedProcess
                    end
          N"N.AttachedDevices =
                    if ~Empty[N"NS.Access]
                        &
                        GrantedAccess
                        then Append[NS.AttachedDevices,
                            <Raddr = NS.Raddr,
                             Access = NS.Access>]
                        else N.AttachedDevices
                    end

    if Empty[N"NS.Access]
        then KernelCalled[SendMessage(OpProcess)]
        else KernelCalled[GrantAccess]
            if GrantedAccess
                then KernelCalled[SendMessage(N.Process)]
                     KernelCalled[SendMessage(OpProcess)]
                else KernelCalled[SendMessage(OpProcess)]
            end
    end
```

UR1:   URProcess request: need Nkcp

        given: RequestedSecLevel: ProcessName
                 Raddr: DeviceAddress

        entry: true

        error on for all (D:URPOwnedDeviceEntry) in URPOwnedDevices:
                    D.Raddr ~= Raddr

        action: let (D:URPOwnedDeviceEntry) in URPOwnedDevices:
                    (D.Raddr = Raddr) in
                if for some (N:NkcpEntry) in CurrentNkcps:
                       (N.Process = RequestedSecLevel)
                    then KernelCall[SendMessage(
                               AddedNkcp[N.Process,D.Raddr],
                               URProcess)]
                  else if Dominates[D.MaxSecLevel,RequestedSecLevel]
                               &
                               Dominates[RequestedSecLevel,D.MinSecLevel]
                               &
                               #Nkcps < #MaxNkcps
                               then KernelCall[CreateProcess(
                                       AddressSpaceSize,
                                       CodeSize,Code)]
                                   if OK
                                      then AddNkcpToSet[RequestedSecLevel]
                                           KernelCall[SendMessage(
                                               AddedNkcp[
                                                   RequestedSecLevel,
                                                   D.Raddr],
                                               URProcess)]
                                      else KernelCall[SendMessage(
                                               CannotAddNkcp[
                                                   RequestedSecLevel,
                                                   D.Raddr],
                                               URProcess)]
                                   end
                          else KernelCall[SendMessage(
                                   CannotAddNkcp[
                                       RequestedSecLevel,
                                       D.Raddr],
                                   URProcess)]
                      end
                end

```
exit:   N"#Nkcps =
                if for all (N:NkcpEntry) in CurrentNkcps:
                    (N.Process ~= RequestedSecLevel)
                &
                Dominates(D.MaxSecLevel,RequestedSecLevel)
                &
                Dominates(RequestedSecLevel,D.MinSecLevel)
                &
                #Nkcps < #MaxNkcps
                &
                CreatedProcess
                then #Nkcps + 1
                else #Nkcps
            end
        N"CurrentNkcps =
                if for all (N:NkcpEntry) in CurrentNkcps:
                    (N.Process ~= RequestedSecLevel)
                &
                Dominates(D.MaxSecLevel,RequestedSecLevel)
                &
                Dominates(RequestedSecLevel,D.MinSecLevel)
                &
                #Nkcps < #MaxNkcps
                &
                CreatedProcess
                then Append(CurrentNkcps,Entry)
                else CurrentNkcps
            end

        if for some (N:NkcpEntry) in CurrentNkcps:
            (N.Process = RequestedSecLevel)
            then KernelCalled(SendMessage(URProcess))
            else if Dominates(D.MaxSecLevel,RequestedSecLevel)
                &
                Dominates(RequestedSecLevel,D.MinSecLevel)
                &
                #Nkcps < #MaxNkcps
                then KernelCalled(CreateProcess)
                        if CreatedProcess
                            then KernelCalled(SendMessage(
                                            URProcess))
                                KernelCalled(SendMessage(
                                            OpProcess))
                                KernelCalled(SendMessage(
                                            URProcess))
                            else KernelCalled(SendMessage(
                                            URProcess))
                    end
                else KernelCalled(SendMessage(URProcess))
            end                                 .
        end
    end
```

197

```
where Entry = <Process = RequestedSecLevel,
              VMs = Empty,
              AttachedDevices = Empty,
              Links = Empty>
```

UR3a: URProcess response to device attachment request (attach succeeded)

```
given:   Raddr: DeviceAddress
         Process: ProcessName
         Laddr: LineAddress

error on ~(for some (A:URPOwnedDeviceEntry) in URPOwnedDevices:
             (A.Raddr = Raddr)
           &
           for some (N:NkcpEntry) in CurrentNkcps:
             (N.Process = Process)
           &
           for some (L:LineEntry) in Lines:
             (L.Laddr = Laddr
              &
              L.State = AttachValidation
              &
              L.CyclePosition = HookingPeripherals
              &
              L.#AwaitingHooks > 0))

action: L.#AwaitingHooks <- L.#AwaitingHooks - 1
        L.Msg <- Concat(L.Msg,Avail[A.Raddr])
        TryNotifyingNkcp(L)
```

```
exit:    N"L.#AwaitingHooks = L.#AwaitingHooks - 1
         N"L.Msg = Concat(L.Msg,Avail(A.Raddr))
         N"L.CyclePosition =
                 if N"L.AwaitingHooks = 0
                    then NotifyingNkcp
                    else L.CyclePosition
                 end
         N"PendingRequests =
                 if N"L.#AwaitingHooks = 0
                    then Append(PendingRequests,Entry)
                    else PendingRequests
                 end

         if N"L.AwaitingHooks = 0
            then if ~L.LineDropped
                    then KernelCalled(SendMessage(
                                  NetworkProcess))
                 end
                 KernelCalled(SendMessage(N.Process))
         end

where Entry  = <Msgid = new(Messageid),
               Kind = NewVM,
               Command = Undefined,
               Responses = (Respondent = NetworkProcess,
                                 Text = nil,
                                 State = NoResponse>)>
```

UR3b: URProcess response to device attachment request (attach
failed)

```
given:  Raddr: DeviceAddress
        Process: ProcessName
        Laddr: LineAddress

error on ~((for some (A:URPOwnedDeviceEntry) in URPOwnedDevices:
            (A.Raddr = Raddr)
          &
          for some (N:NkcpEntry) in CurrentNkcps:
            (N.Process = Process)
          &
          for some (L:LineEntry) in Lines:
            (L.Laddr = Laddr
            &
            L.State = AttachValidation
            &
            L.CyclePosition = HookingPeripherals
            &
            L.#AwaitingHooks > 0))

action: L.#AwaitingHooks <- L.#AwaitingHooks - 1
        L.Msg <- Concat(L.Msg,Unavail(A.Raddr))
        TryNotifyingNkcp(L)
```

201

```
exit:    N"L.#AwaitingHooks = L.#AwaitingHooks - 1
         N"L.Msg = Concat(L.Msg,Unavail(A.Raddr))
         N"L.CyclePosition =
                 if N"L.AwaitingHooks = 0
                 then NotifyingNkcp
                 else L.CyclePosition
             end
         N"PendingRequests =
                 if N"L.#AwaitingHooks = 0
                 then Append(PendingRequests,Entry)
                 else PendingRequests
             end

         if N"L.AwaitingHooks = 0
             then if ~L.LineDropped
                 then KernelCalled(SendMessage(
                                 NetworkProcess))

             end
             KernelCalled(SendMessage(N.Process))

         end

    where Entry = <MsgId = new(MessageId),
                   Kind = NewVM,
                   Command = Undefined,
                   Responses = (Respondent = NetworkProcess,
                                Text = nil,
                                State = NoResponse>)>
```

NKCP3: Drop User

        given: Process: ProcessName
                VM: VirtualMachineName
                Laddr: LineAddress

        error on ~(for some (N:NkcpEntry) in CurrentNkcps:
                    (N.Process = Process
                      &
                     for some (V:VMEntry) in N.VMs:
                        (V.VMName = VM
                          &
                         for some (U:LineAddress) in V.Users:
                            U = Laddr))
                 &
                 for some (L:LineEntry) in Lines:
                    (L.Laddr = Laddr
                      &
                     L.State = Attached
                      &
                     L.Connection = Dial
                      &
                     L.AttachedVM = VM))

        action: L.State <- Free
                L.CyclePosition <- ReEnablePending
                KernelCall(SendMessage(ClearLine[L.Laddr],
                        NetworkProcess))
                KernelCall(SendMessage(Dropped[VM,U],
                        OpProcess))
                N.V.Users <- Remove[N.V.Users,U]
                KernelCall(SendMessage(
                        SystemResourceUse[N.Process,VM],
                        AcntProcess))

        exit: N"L.State = Free
                N"L.CyclePosition = ReEnablePending
                N"N.V.Users = Remove[N.V.Users,U]
                N"PendingRequests = Append[PendingRequests,
                        <MsgId = new[MessageId],
                         Kind = ClearLine,
                         Command = Undefined,
                         Responses = (Respondent = NetworkProcess,
                                        Text = nil,
                                        State = NoResponse>]>]

                KernelCalled(SendMessage(NetworkProcess))
                KernelCalled(SendMessage(OpProcess))
                KernelCalled(SendMessage(AcntProcess))

                                    203

NTWK3:   Link password received

        given: Process: ProcessName
                 Password: string
                 Requester: VirtualMachineName
                 Laddr: LineAddress
                 User: VirtualMachineName
                 MiniDisk: MiniDiskName
                 ReqAccess: LinkAccess

        entry: for some (L:LineEntry) in Lines:
                     (L.Laddr = Laddr
                        &
                      L.State = Attached
                        &
                      L.CyclePosition = ReadLinkPassword)

        error on ~(for some (N:NkcpEntry) in CurrentNkcps:
                     (N.Process = Process))

        action: KernelCall(SendMessage(
                        ReDirectLine(L.Laddr,N.Process),
                        NetworkProcess))
                L.CyclePosition <- Attached
                let (D:DirectoryEntry) in UserDirectory:
                    (D.UserId = User) in
                if D.LinkPassword = Password
                    then let (M:MiniDiskEntry) in MiniDisks:
                             (M.MDName = MiniDisk) in
                         let (V:SharedVolumeEntry) in SharedVolumes:
                             (V.Volume = M.ContainingVolume) in
                         if V.State = Mounted
                            then let (S:SharableDriveEntry)
                                            in SharableDrives:
                                    (S.Raddr = V.MountedDevice) in
                                 if S.State = AttachedToSystem
                                    then /* WHEW! everything is OK
                                            to actually process the
                                            link request */
                                         let (A:ACLEntry)
                                                 in M.AccessControlList:
                                             (A.User = Requester) in
                                         ExamineLinkAccess(N,M,
                                                 ReqAccess)
                                    else NoLink(DeviceNotReady)
                                 end
                            else NoLink(VolumeNotMounted)
                         end
                    else NoLink(IllegalPassword)
                end

                                204

```
macro NoLink [Reason] =

        KernelCall [SendMessage(
                        CannotLink [
                                Requester, User,
                                Laddr, MiniDisk,
                                Reason],
                    N.Process)]
```

```
macro ExamineLinkAccess [N:NkcpEntry in CurrentNkcps,                    )
                         M:MiniDiskEntry in MiniDisks,
                         Access:LinkAccess] =

       case Access:LinkAccess of
            R:
                 if for all (C:ProcessLinkEntry) in M.CurrentLinks:
                         Write ~inset C.Access
                      then Link[(Read)]
                      else NoLink[PreviousWriteLink]
                 end

            RR:
               Link[(Read)]

            W:
               if Write inset A.Access
                  then if Empty(M.CurrentLinks)
                          then Link[(Write)]
                          else NoLink[PreviousLink]
                       end
                  else NoLink[NoWritePermission]
               end

            WR:
               if Write inset A.Access
                  then if Empty(M.CurrentLinks)
                          then Link[(Write)]
                          else Link[(Read)]
                       end
                  else /* choices:     Link[(Read)]
                                       NoLink[NoWritePermission]    */
                       NoLink[NoWritePermission]
               end

            M:
               if Write inset A.Access
                  then if for all (C:ProcessLinkEntry)
                                        in M.CurrentLinks:
                               Write ~inset C.Access
                          then Link[(Write)]
                          else NoLink[PreviousWriteLink]
                       end
                  else NoLink[NoWritePermission]
               end
```

206

```
MR:
    if Write inset A.Access
        then if for all (C:ProcessLinkEntry)
                            in M.CurrentLinks:
                    Write ~inset C.Access
                then Link[(Write)]
                else Link[(Read)]
            end
        else /* choices:      Link[(Read)]
                              NoLink[NoWritePermission]   */
            NoLink[NoWritePermission]
    end

MW:
    if Write inset A.Access
        then Link[(Write)]
        else NoLink[NoWritePermission]
    end
end
```

```
macro Link(Access:set of AccessModes) =

        /* given: N:NkcpEntry in CurrentNkcps
                  M:MiniDiskEntry in MiniDisks   */

    if for some (C:ProcessLinkEntry) in M.CurrentLinks:
           C.Process = N.Process
        then /* process already has a link to this minidisk:
                 increase rights if necessary */
           let (NC:MDLinkEntry) in N.Links:
               (NC.MDName = M.MDName) in
            if Access ~= C.Access
               then if Read inset Access
                        &
                        Read ~inset C.Access
                        then /* in first cut of system,
                                 all links include read
                                 permission: should never
                                 get here */
                             KernelCall(GrantAccess(
                                N.Process,M.MDName, (Read)))
                             if OK
                                then C.Access <- Append[
                                                 C.Access,Read]
                                     NC.Access <- Append[
                                                 NC.Access,Read]
                                else NoLink(NotOK)
                             end
                  end
                  if Write inset Access
                     &
                     Write ~inset C.Access
                     then KernelCall(GrantAccess(
                                N.Process,M.MDName, (Write)))
                             if OK
                                then C.Access <- Append[
                                                 C.Access,Write]
                                     NC.Access <- Append[
                                                 NC.Access,Write]
                                else NoLink(NotOK)
                             end
                  end
            else /* not necessary to increase rights
                     then why did NKCP ask for it?? */
        end
```

208

```
        else KernelCall(GrantAccess(
                    N.Process,M.MDName,Access))
            if OK
                then M.CurrentLinks <- Append[
                        M.CurrentLinks,
                        <Process = N.Process,
                         Access = Access>]
                     N.Links <- Append[
                        N.Links,
                        <MDName = M.MDName,
                         Access = Access>]
                else NoLink(NotOK)
            end
    end
    KernelCall(SendMessage(
            LinkStatus(N),
            N.Process))
```

NKCP4:   Link (with password)

given: Process: ProcessName
         Requester: VirtualMachineName
         Laddr: LineAddress
         User: VirtualMachineName
         MiniDisk: MiniDiskName
         ReqAccess: LinkAccess

error on ShuttingDown
           |
         Empty(ReqAccess)

action: if for some (N:NkcpEntry) in CurrentNkcps:
              (N.Process = Process
               &
               for some (E:VMEntry) in N.VMs:
                   (E.VMName = Requester))
         &
         for some (L:LineEntry) in Lines:
             (L.Laddr = Laddr
              &
              L.State = Attached
              &
              L.CyclePosition = Attached
              &
              L.RequestedSecLevel = Process
              &
              L.AttachedVM = Requester)
         &
         for some (D:DirectoryEntry) in UserDirectory:
             (D.UserId = User
              &
              for some (K:MDLinkEntry) in D.Links:
                  (K.MDName = MiniDisk))
         &
         for some (M:MiniDiskEntry) in MiniDisks:
             (M.MDName = MiniDisk)
             &

210

```
            for some (A:ACLEntry) in M.AccessControlList:
                (A.User = Requester
                &
                Dominates(Process,M.SecLevel)
                &
                (WriteAccessRequested =>
                        Process = M.SecLevel))
            then /* it's legal: now check
                    for resource availability */
                if for some (V:SharedVolumeEntry)
                                        in SharedVolumes:
                    (V.Volume = M.ContainingVolume
                     &
                     V.State = Mounted
                     &
                     for some (S:SharableDriveEntry)
                                        in SharableDrives:
                        (S.Raddr = V.MountedDevice
                         &
                         S.State = AttachedToSystem))
                then /* resources are available:
                        get the link share password */
                    KernelCall(SendMessage(
                        ReDirectLine(L.Laddr,AuthProcess),
                        NetworkProcess))
                    KernelCall(SendMessage(
                            ReadLinkPassword(L.Laddr),
                            NetworkProcess))
                    L.CyclePosition <- ReadLinkPassword
                else /* resources are not available */
                    NoLink(ResourcesNotAvailable)
            end
        else /* not a legal request */
            NoLink(IllegalRequest)
    end

exit:    N"L.CyclePosition =
                if Legal
                    &
                    ResourcesAvailable
                    then ReadLinkPassword
                    else L.CyclePosition
                end
        N"PendingRequests =
                if Legal
                    &
                    ResourcesAvailable
                    then Append(PendingRequests,Entry)
                    else PendingRequests
                end
```

211

```
where Entry  = <MsgId = new(MessageId),
                Kind = WriteAndReadLine,
                Command = Undefined,
                Responses = (Respondent = NetworkProcess,
                             Text = nil,
                             State = NoResponse>)>

KernelCalled(SendMessage(NetworkProcess))
```

212

NKCP5, NKCP6: Detach nonsharable device (request from process), and response (from process) to relinquish device request from authorization process

```
given:  Raddr: DeviceAddress
        Process: ProcessName
        User: VirtualMachineName

error on ~(for some (N:NkcpEntry) in CurrentNkcps:
            (N.Process = Process
            &
            for some (VM:VMEntry) in N.VMs:
                (VM.VMName = User)
            &
            for some (A:AttachedDeviceEntry)
                                    in N.AttachedDevices:
                (A.Raddr = Raddr))
         &
         for some (NS:NonsharableDriveEntry)
                                    in NonsharableDrives:
            (NS.Raddr = Raddr
            &
            NS.AttachedProcess = Process))

action: KernelCall(ReleaseDevice(NS.Raddr))
        if DeviceReleased
            then NS.State <- Available
                N.AttachedDevices <- Remove(
                                        N.AttachedDevices,
                                        A)
                KernelCall(SendMessage(
                        Available(NS.Raddr),
                        OpProcess))
                KernelCall(SendMessage(
                        DeviceUse(User,NS.Raddr),
                        AcntProcess))
            else KernelCall(SendMessage(
                        NotDetached(NS.Raddr,Process),
                        OpProcess))
                KernelCall(SendMessage(
                        NotDetached(NS.Raddr),
                        NS.AttachedProcess))
        end
```

```
exit:    N"NS.State =
                if DeviceReleased
                    then Available
                    else NS.State
                end
         N"N.AttachedDevices =
                if DeviceReleased
                    then Remove(N.AttachedDevices,A)
                    else N.AttachedDevices
                end

         KernelCalled(ReleaseDevice(NS.Raddr))
         KernelCalled(SendMessage(NoProcess))
         if DeviceReleased
             then KernelCalled(SendMessage(AcntProcess))
             else KernelCalled(SendMessage(NS.AttachedProcess))
         end
```

214

NKCP7: Purge NKCP

given: Process: ProcessName

entry: true

error on ~(for some (N:NkcpEntry) in CurrentNkcps:
            (N.Process = Process
             &
             Empty[N.VMs]
             &
             Empty[N.AttachedDevices]
             &
             Empty[N.Links]))

exit: N"CurrentNkcps = Remove(CurrentNkcps,N]

215

KERN1: message from Kernel, re shared device availability


```
given:  Raddr: DeviceAddress
        Volume: VolumeId
        CurrentStatus: SharableDriveStatus

error on ~((for some (S:SharableDriveEntry) in SharableDrives:
                (S.Raddr = Raddr
                 &
                (CurrentStatus = AttachedToSystem =>
                    (for some (V:SharedVolumeEntry)
                                    in SharedVolumes:
                        (V.Volume = Volume)))))

action:  if S.State = AttachedToSystem
            then if CurrentStatus = AttachedToSystem
                then if Volume ~= S.MountedVolume
                    then if for some (M:MiniDiskEntry)
                                    in MiniDisks:
                                    (M.Containing Volume =
                                             S.MountedVolume
                                     &
                                    ~Empty(M.CurrentLinks))
                            then /* security error:
                                     wrong volume
                                     in use */
                                 error
                            else /* old volume not in
                                    use: OK to switch */
                            let (Vold:SharedVolumeEntry)
                                    in SharedVolumes:
                                    (Vold.Volume =
                                             S.MountedVolume
                                     &
                                    Vold.State = Mounted
                                     &
                                    Vold.MountedDevice =
                                             S.Raddr) in
                            let (Vnew:SharedVolumeEntry)
                                    in SharedVolumes:
                                    (Vnew.Volume = Volume) in
```


216

```
                                        If Dominates[
                                            S.SecLevel,
                                            Vnew.SecLevel]
                                            then S.MountedVolume <-
                                                    Vnew.Volume
                                                Vold.State <-
                                                        NotMounted
                                                Vold.MountedDevice <-
                                                        nil
                                                Vnew.State <-
                                                        Mounted
                                                Vnew.MountedDevice <-
                                                        S.Raddr
                                                KernelCall[
                                                    DriveMatchesVolume(
                                                        Raddr,Volume)]
                                            else KernelCall[
                                        DriveDoesNotMatchVolume(
                                                        Raddr,Volume)]

                                        end
                                        end
                            else /* OK: same volume,
                                        same state */
                                    KernelCall[DriveMatchesVolume(
                                        Raddr,Volume)]

                    end
            else /* old state = attached to system,
                    new state = not attached;
                    update table entries */
                    let (V:SharedVolumeEntry)
                                    in SharedVolumes:
                        (V.Volume = S.MountedVolume
                            &
                            V.State = Mounted
                            &
                            V.MountedDevice = S.Raddr) in
                        V.State <- NotMounted
                        V.MountedDevice <- nil
                        S.MountedVolume <- nil
                        S.State <- CurrentStatus

        end
```

217

```
                    else /* old state = not attached to system */
                       if CurrentStatus = AttachedToSystem
                          then let (V:SharedVolumeEntry)
                                   in SharedVolumes:
                                   (V.Volume = Volume) in
                             if Dominates[S.SecLevel,V.SecLevel]
                                then S.State <- AttachedToSystem
                                     S.MountedVolume <- V.Volume
                                     V.State <- Mounted
                                     V.MountedDevice <- S.Raddr
                                     KernelCall[
                                        DriveMatchesVolume(
                                           Raddr,Volume)]
                                else KernelCall[
                                        DriveDoesNotMatchVolume(
                                           Raddr,Volume)]
                       end
                    else S.State <- CurrentStatus
                 end
     end
```

218

Accounting Process
Semi-Formal Description

This section contains a semi-formal description of the Accounting
Process of KVM/370.

Data Types

primitive types and structuring mechanisms:

        boolean [unordered, two elements: true, false]
        string [unbounded, predefined string of length zero: nil]

        set   [of any type, predefined empty set: nil]
        record [field list]

undefined types:

        VirtualMachineName
        ProcessName
        MessageId

AccountingRecord:
        record
            User: VirtualMachineName
            Postings: set of string
        end

Data Structures

Accounting: set of AccountingRecord

Initial Conditions

Empty(Accounting)

Invariant Assertions

for all (A1,A2:AccountingRecord) in Accounting:
    (A1.User = A2.User => A1 = A2)
&
for all (A:AccountingRecord) in Accounting:
    (~Empty(A.Postings))

Global Macros / Functions

primitive macros / functions:

        Append(set,entry)

220

MsgAuth: process

       /* subdriver of AcntProcess,
             handling messages from AuthProcess */

       given: Text: string

       entry: just received message, Source = AuthProcess

       action: error on MsgName(Text) ~inset   (SystemResourceUse,
                                         DeviceUse)
           AUTH1
end MsgAuth

221

MsgOp: process

    /* subdriver of AcntProcess,
        handling messages from OpProcess */

    given: MsgId: MessageId
           Text: string

    entry: just received message, Source = OpProcess

    action: error on MsgName[Text] ~inset (ACNT←PUNCH,
                         SHUTDOWN)
        OP1

end MsgOp

```
AcntDriver:  process

        /* driver of AcntProcess */

        case HowWeGotHere of
            ExternalInterrupt:
                case InterruptSubType of
                    Message:
                        case Source of
                            OpProcess: MsgOp
                            Auth:      MsgAuth
                            other:     /* anybody else talk
                                          with AcntProcess? */

                    end

                    other:
                        /* any other external interrupts? */
                end

            other:
                /* any other important interrupt classes? */
        end

        KernelCall [ReceiveInterrupts]
        KernelCall [ReleaseCPU]

end AcntDriver
```

223

AUTH1:   Accounting record from Nkcp via Authorization Process:
System Resource Use or Device Use

    given: User: VirtualMachineName
             Text: string

    entry: true /* user id has been validated by AuthProcess
                      prior to the sending of this message */

    action: if for some (A:AccountingRecord) in Accounting:
                    A.User = User
                 then /* user id already exists in data base */
                     A.Postings <- Append[A.Postings,Text]
                 else /* user id does not exist in data base:
                         add it */
                     Accounting <- Append[Accounting,
                            <User = User,
                             Postings = Append[nil,Text]>]
             end

    exit: for some (A:AccountingRecord) in Accounting:
                 (A.User = User
                  &
                  for some (S:string) in A.Postings:
                      S = Text)

224

OP1:   Operator command to re-initialize the accounting data base

      entry: true

      exit: N"Accounting = nil

          KernelCalled(SendMessage(OpProcess))

Updater Process
Semi-Formal Description

This section contains a semi-formal description of the Updater
Process of KVM/370.

Data Types

primitive types and structuring mechanisms:

        boolean [unordered, two elements: true, false]
        string   [unbounded, predefined string of length zero: nil]
        integer subrange

        scalar   [ordered element list]
        set   [of any type, predefined empty set: nil]
        record   [field list]

undefined types:

        DeviceAddress
        LineAddress
        ProcessName
        VirtualMachineName
        VolumeId

undefined functions / macros:

        Dominates
        DeviceType
        #Cylinders

AccessModes: scalar(
        Read,
        Write)

PossibleEntries: scalar(
        Paging,
        Spooling,
        MiniDisk,
        Unknown,
        System)

DirectoryEntry:
        record
                Userid: VirtualMachineName
                LogonPassword: string
                DialPassword: string
                LinkPassword: string
                MaxSecLevel: ProcessName
                MinSecLevel: ProcessName
                DedicatedDevices: set of DedicatedDeviceEntry
                Links: set of MDLinkEntry
                IplDefined: boolean
                AccessPasswords: set of AccessPasswordEntry
        end

LineEntry:
        record
                Laddr: LineAddress
                MinSecLevel: ProcessName
                MaxSecLevel: ProcessName
        end

AccessPasswordEntry:
        record
                SecLevel: ProcessName
                Password: string
        end

DedicatedDeviceEntry:
        record
                Raddr: DeviceAddress
                VolSecLevel: ProcessName
                Access: set of AccessModes
        end

MDLinkEntry: •
        record
                MDName: MiniDiskName
                Access: set of AccessModes
        end

URPOwnedDeviceEntry:
        record
                Raddr: DeviceAddress
                MaxSecLevel: ProcessName
                MinSecLevel: ProcessName
        end

227

NonsharableDriveEntry:
        record
                Raddr: DeviceAddress
                MaxSecLevel: ProcessName
                MinSecLevel: ProcessName
        end

SharableDriveEntry:
        record
                Raddr: DeviceAddress
                SecLevel: ProcessName
        end

SharedVolumeEntry:
        record
                Volume: VolumeId
                SecLevel: ProcessName
                Map: set of CylMap
        end

CylMap:
        record
                Cylinders: (1..#MaxCylinders,
                            1..#MaxCylinders)
                Category: PossibleEntries
        end

MiniDiskEntry:
        record
                MDName: MiniDiskName
                ContainingVolume: VolumeId
                Cylinders: (1..#MaxCylinders,
                            1..#MaxCylinders)
                SecLevel: ProcessName
                AccessControlList: set of ACLEntry
        end

ACLEntry:
        record
                User: VirtualMachineName
                Access: set of AccessModes
        end

Data Structures

constant #MaxCylinders: positive integer

URPOwnedDevices: set of URPOwnedDeviceEntry

NonsharableDrives: set of NonsharableDriveEntry

SharableDrives: set of SharableDriveEntry

SharedVolumes: set of SharedVolumeEntry

MiniDisks: set of MiniDiskEntry

Lines: set of LineEntry

UserDirectory: set of DirectoryEntry

```
entry:   true

exit:    DistinctDeviceAddresses
         &
         LegalUserDirectory
         &
         LegalLines
         &
         LegalMiniDisks
         &
         LegalSharedVolumes
         &
         LegalSharableDrives
         &
         LegalNonsharableDrives
         &
         LegalURPOwnedDevices


DistinctDeviceAddresses =

        for all (U:URPOwnedDeviceEntry) in URPOwnedDevices:
            (for all (NS:NonsharableDriveEntry) in NonsharableDrives:
                (U.Raddr ~= NS.Raddr)
            &
            for all (S:SharableDriveEntry) in SharableDrives:
                (U.Raddr ~= S.Raddr))
        &
        for all (NS:NonsharableDriveEntry) in NonsharableDrives:
            for all (S:SharableDriveEntry) in SharableDrives:
                NS.Raddr ~= S.Raddr
```

230

LegalUserDirectory =

    for all (U1,U2:DirectoryEntry) in UserDirectory:
      (U1.Userid = U2.Userid => U1 = U2)
    &

    for all (U:DirectoryEntry) in UserDirectory:
      (Dominates(U.MaxSecLevel,U.MinSecLevel)
      &

(1.1)   LegalDedicatedDevices(U)
        &

(1.2)   LegalLinks(U)
        &

(1.3)   LegalAccessPasswords(U))

231

```
LegalDedicatedDevices(U:DirectoryEntry) =

        for all (E1,E2:DedicatedDeviceEntry) in U.DedicatedDevices:
        (E1.Raddr = E2.Raddr => E1 = E2)
    &
        for all (E:DedicatedDeviceEntry) in U.DedicatedDevices:
            for some (D:URPOwnedDeviceEntry) in URPOwnedDevices:
            (D.Raddr = E.Raddr
             &
             (DeviceType(E.Raddr) = Reader =>
                     (E.VolSecLevel = nil
                      &
                      E.Access = {Read}))
             &
             (DeviceType(E.Raddr) inset {Printer, Punch} =>
                     (E.VolSecLevel = nil
                      &
                      E.Access = {Write}))
             &
             (DeviceType(Raddr) = TapeDrive =>
                     (Dominates(D.MaxSecLevel,E.VolSecLevel)
                      &
                      Dominates(E.VolSecLevel,D.MinSecLevel)
                      &
                      Dominates(U.MaxSecLevel,E.VolSecLevel)
                      &
                      ~Empty(E.Access)))
            )
        xor
        for some (D:NonsharableDriveEntry) in NonsharableDrives:
            (D.Raddr = Raddr
             &
             Dominates(D.MaxSecLevel,E.VolSecLevel)
             &
             Dominates(E.VolSecLevel,D.MinSecLevel)
             &
             Dominates(U.MaxSecLevel,E.VolSecLevel)
             &
             ~Empty(E.Access))
```

232

```
LegalLinks(U:DirectoryEntry) =

        for all (L1,L2:MDLinkEntry) in U.Links:
            (L1.MDName = L2.MDName => L1 = L2)
        &
        for all (L:MDLinkEntry) in U.Links:
            for some (M:MiniDiskEntry) in MiniDisks:
                (M.MDName = L.MDName
                 &
                 for some (A:ACLEntry) in M.AccessControlList:
                    (A.User = U.UserId
                     &
                     for all (AM:AccessModes) in L.Access:
                        AM inset A.Access)
                 &
                 Dominates(U.MaxSecLevel,M.SecLevel))
        &
        ~Empty(L.Access)
```

233

```
LegalAccessPasswords(U:DirectoryEntry) =

        for all (A1,A2:AccessPasswordEntry) in U.AccessPasswords:
            (A1.SecLevel = A2.SecLevel => A1 = A2)
        &
        for all (A:AccessPasswordEntry) in U.AccessPasswords:
            (Dominates(U.MaxSecLevel,A.SecLevel)
              &
              Dominates(A.SecLevel,U.MinSecLevel))
```

LegalLines =

        for all (L1,L2:LineEntry) in Lines:
          (L1.Laddr = L2.Laddr => L1 = L2)
        &
        for all (L:LineEntry) in Lines:
          Dominates(L.MaxSecLevel,L.MinSecLevel)

235

LegalMiniDisks =

```
        for all (M1,M2:MiniDiskEntry) in MiniDisks:
            (M1.MDName = M2.MDName => M1 = M2)
        &
        for all (M:MiniDiskEntry) in MiniDisks:
(3.1)       (LegalContainingVolume(M)
            &
            M.Cylinders.1 < M.Cylinders.2
            &
            M.Cylinders.2 <= #Cylinders(M.ContainingVolume)
            &
            M.Cylinders.1 < #Cylinders(M.ContainingVolume)
            &
(3.2)       LegalAccessControlList(M))
```

236

```
LegalContainingVolume(M:MiniDiskEntry) =
        for some (S:SharedVolumeEntry) in SharedVolumes:
            (S.Volume = M.ContainingVolume
             &
            Dominates(S.SecLevel,M.SecLevel)))
```

237

```
LegalAccessControlList(M:MiniDiskEntry) =

        for all (A1,A2:ACLEntry) in M.AccessControlList:
           (A1.User = A2.User => A1 = A2)
        &
        for all (A:ACLEntry) in M.AccessControlList:
           (for some (D:DirectoryEntry) in UserDirectory:
               (D.UserId = A.User)
            &
            ~Empty(A.Access))
```

238

LegalSharedVolumes =

   for all (S1,S2:SharedVolumeEntry) in SharedVolumes:
    (S1.Volume = S2.Volume => S1 = S2)
   &
   for all (S:SharedVolumeEntry) in SharedVolumes:
(4.1)   (LegalMap(S))

```
LegalMap(S:SharedVolumeEntry) =

        /* non-overlap */
        for all (M1,M2:CylMap) in S.Map:
            (M1.Cylinders.1 > M2.Cylinders.2
             or
             M1.Cylinders.2 < M2.Cylinders.1)
        &
        for all (M:CylMap) in S.Map:
            (/* each entry non-empty */
             M.Cylinders.2 > M.Cylinders.1
             &
             /* no cylinders unaccounted for */
             M.Cylinders.2 ~= #Cylinders(S) =>
                    for some (M1:CylMap) in S.Maps:
                        M1.Cylinders.1 = M.Cylinders.2 + 1)
        &
        for some (M:CylMap) in S.Map:
            (M.Cylinders.1 = 1)
        &
        /* each MiniDisk actually logged */
        for all (M:CylMap) in S.Map:
            (M.Category = MiniDisk =>
                    for some (MD:MiniDiskEntry) in MiniDisks:
                        (MD.ContainingVolume
                         &
                         MD.Cylinders = M.Cylinders))
```

240

LegalSharableDrives =

    for all (SD1,SD2:SharableDriveEntry) in SharableDrives:
        SD1.Raddr = SD2.Raddr => SD1 = SD2

LegalNonsharableDrives =

      for all (NS1,NS2:NonsharableDriveEntry) in NonsharableDrives:
        (NS1.Raddr = NS2.Raddr => NS1 = NS2)
     &δ
      for all (NS:NonsharableDriveEntry) in NonsharableDrives:
        Dominates(NS.MaxSecLevel,NS.MinSecLevel)

LegalURPOwnedDevices =

      for all (U1,U2:URPOwnedDeviceEntry) in URPOwnedDevices:
        (U1.Raddr = U2.Raddr => U1 = U2)
      &
      for all (U:URPOwnedDeviceEntry) in URPOwnedDevices:
        (Dominates(U.MaxSecLevel,U.MinSecLevel)
         &
        DeviceType(Raddr) inset   (Reader,
                               Printer,
                               Punch,
                               TapeDrive))

243